SOFTWARE ARCHITECTURE FOR PARALLEL PROCESSORS

W. C. Cave† & R.E. Wassmer† - August 17, 2016

BACKGROUND

In the early Renaissance, artists sketched buildings that represented their imagined plans. Their renderings contained no measurements. Builders were expected to follow the illustrations and work out the details. There were no engineering principles or drawings. Even now, one need not go to engineering school to design a dog house. It needs no drawings or measurements.

As buildings became large and more complex, the artist's approach was forced to change, eventually becoming known as architectural engineering. By careful design of many levels of detail, an engineering process has evolved for solving construction problems - before the construction begins. Improvements are implemented on a drawing board and in written specifications, avoiding costly mistakes, huge delays and corresponding cost overruns.

In the ACM article, *The Emperor with No Clothes*, [1], Henry Ledgard quoted W. Edwards Deming who stated "If you can't measure it, you can't improve it," see [8] The same point was made by David Parnas, [2], 10 years earlier: "Without measures from repeatable experiments, software is not a science." Although an initiator of Computer Science curricula, Parnas said: "most CS PhDs are not scientists; they neither understand nor apply the methods of experimental science." Ledgard and Parnas are highly knowledgeable in computer languages.

At the top is Grace Hopper, who wrote the first compiler while at Univac in 1952. In 1959, after the CODASYL conference initiated the development of COBOL, Hopper's group at Univac spearheaded the language design based upon her own FLOW-MATIC language, see Beyer, [3]. Hopper stated that programs should be written in a language close to English rather than those close to machine code or assembler, see Ledgard [9]. This was captured in the new language, COBOL, which would become the most ubiquitous data system language to date. Hopper went on to develop CMS-2, a language for the U.S. Navy, adding math and scientific facilities to COBOL. CMS-2 contained the same hierarchical data and hierarchical instruction syntax that contributes huge productivity gains and applies directly to parallel processing.

But now the time has come to apply engineering science to software. The dog house approach to memorizing snippets of code does not provide the knowledge required to design complex software systems and simulations. To meet pressing requirements for increased speed, computers have gone parallel with large numbers of processors on each chip, and many chips on a board. Today's applications are the equivalent of skyscrapers. Moving the software field forward into an engineering discipline is an obvious necessity.

INTRODUCTION

This paper describes an engineering approach to software based on a theory that is an extension of mathematics. Achieving speed in development and support - as well as at run time - requires a solid science foundation. This implies conducting repeatable laboratory experiments to ensure that theories are adopted based on fair comparison of carefully measured results.

† Visual Software International - www.VisiSoft.com

When designing a complex system, one breaks it into modules to gain architectural independence of various operations. As individual modules become more complex, they are layered into hierarchies. Without a clear hierarchy of modules, it is difficult to understand the bottom layer of detail - where careful design decisions are made. When moving from scalars into complex state spaces, one must be able to understand the hierarchies of a space in order to design the "best" algorithms, i.e., those that are fast and easy to understand. This requires visualization of hierarchies that directly represent the space.

When dealing with complex software module architectures, one must be able to visualize the hierarchies of functions that simplify and speed operations. This implies the ability to observe directly how complex data structures are shared with complex sets of instructions. In VSE, this is accomplished by separation of data from instructions, also known as the "Separation Principle," [4]. This separation has existed in computer hardware design since the RISC chip. Having separated instructions (processes) from data (memory resources), one can represent software architectures using engineering drawings. The intent of this paper is to show how architecture is as important to software as it is to the design and construction of physical structures and devices.

APPLICATION SPACE ARCHITECTURE - THE ISA FOR PARALLEL PROCESSORS

The first *stored program* computer (the MANIAC) was used to design the first hydrogen bomb. It required solving a much larger system of equations. This required a relatively huge increase in memory to store the data as well as instructions required to meet the speed constraints. The MANIAC design was based on von Neumann's Instruction Set Architecture (ISA), developed to support a broad base of applications. These first computer designs made it clear that memory size was the major factor in achieving speed.

To solve the parallel processor software technology problem, one must take maximum advantage of the inherent parallelism in each particular application to minimize running time. This requires designing the best spaces in which to map the inherent parallelism to solve the problem. As shown in Figure 2, this requires mapping the application requirements into a software space that is defined by the software development environment. This environment must support ease of translation of the application requirements into a software language. It must also map into the seven layer model for software depicted in Figure 1.

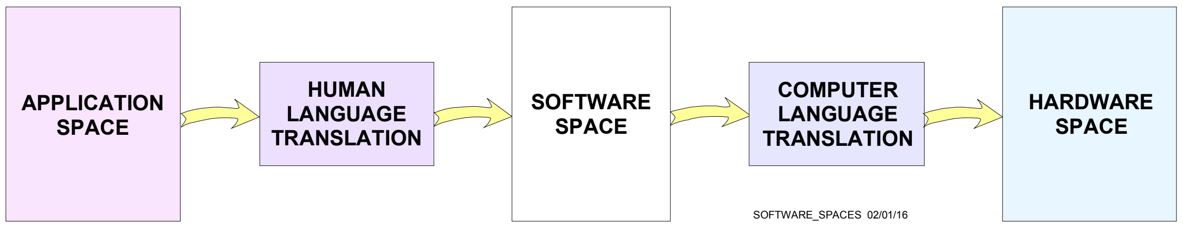


Figure 1. The ASA spaces of software design.

Software Spaces - An Extension of Mathematics

Having defined an application space, one can focus on the design of a software space that meets the requirements and makes optimal use of the hardware space. Demanding applications require the breakthrough speeds of advanced parallel processors. This implies software spaces that maximize the simplicity and understandability of the software design while meeting the application requirements, particularly the speed and accuracy constraints, on the latest parallel processor hardware technology.

Software Spaces require an extension of mathematics beyond the concepts of *State Space* used in control theory. Once used, the extensions become obvious. The critical differences are the use of English-like words to represent binary numbers, the use of deep complex hierarchies representing the necessary data space descriptions, and English-like statements defining operations that support complex decision processes. These are further defined below.

With a well designed software space, large parallel processor software systems can be built and modified easily by people with application expertise. Their knowledge is critical to decomposing systems with inherent parallelism into easily understood hierarchies - at both the architectural and language levels. Such knowledge is required to linearize complexity while maximizing run-time speed.

To achieve these goals, the ASA must support critical properties, a sampling of which follows.

- Spatial Selection - Application systems must be decomposed into a hierarchy of spaces that simplify understanding of the required transformations. These spaces must be organized around the system's events, and effects of interactions of the subsystems. For complex systems, this requires the deep knowledge of an application expert.
- Spatial Distribution - Independent elements of systems typically share information that is copied and available directly to each element. Saving memory is an abstraction that causes delays - waiting for access to shared spatial memory.
- Spatial and Temporal Synchronization - Independent elements of systems that share information are synchronized based on detailed application event dynamics. These event dynamics are typically only understood by application experts. Facilities must exist so these experts can easily represent the required synchronization directly.
- Independence of Modules - Systems must be decomposed into maximally independent modules. This is apparent with interactive systems, and easily implemented following the physical dynamics of the independent elements of the application directly.
- Understandability of Modules - Modules must be easily understood by other than the original author. This implies following the principles described by Shannon in his work on Theory of Communications and by famous language designer Grace Hopper. For example, in software:
  - Names identifying spatial vectors, subvectors and elements must maximize clarity and understanding of the required transformations.
  - Functions must be specified explicitly as opposed to implicitly.
- Hierarchy of Modules - Module decomposition must follow a hierarchy that relates directly to the run-time operation of the system.

- Orthogonality of Software Design - Software design of a non-sequential system is typically orthogonal to the organization of its functional requirements. However, a good software design typically follows directly from the physical dynamics of the application. This is particularly important with interactive systems.

Resources can be dedicated to a single process, shared by more than one process, use different areas of memory, be shared across tasks or simulations, or be shared across multiple computers. Resources are also used to define external files, including fixed or variable length records, sequential or direct file access methods, and binary or printable text files. These resource types are defined below.

KEY CONCEPTS FROM PROVEN THEORIES

Software Spaces For Parallel Processing

The development environment that people share is critical, especially the tools they use to create and communicate their parts of the design. The best examples of such tools are Computer-Aided Design (CAD) systems. This is stated emphatically by Broy, [5] and Poore, [6], both describing the need for an engineering approach supported by a CAD environment for developing software. The companion to these papers, [7], describes the contribution of key concepts from prior proven theories, and introduces *VisiSoft*®, a CAD environment that makes it easy to implement the desired concepts.

To simplify software development on parallel processors, one must be able to map the inherent parallelism in an application into a software architecture such that processes can run concurrently. This implies creating processes that are independent, i.e., they share no data directly. To determine the independence of processes, designers must be able to easily see which processes share what data (memory resources). This can only be done when the following *Critical Software Architecture Requirements* are met:

- Data is organized into a minimum number of structures shared between processes;
- Data structures can be organized into the deep hierarchies required to represent the best spaces to implement problem solutions;
- Designers can easily determine which processes share what data so they can assure their independence properties.

The above requirements are best met when data is separated from instructions at the language level. This separation supports the design of a data language for organizing large data structures using deep hierarchies. It also supports design of an instruction language for building hierarchies of rule structures. Both looping and complex IF ... THEN ... ELSE statements are then flattened. What is known as *Waterfall* or *Fall through* code is gone (without GOTOs). These properties dramatically simplify design of the best data spaces, and concurrently, the design of complex algorithms. Both lead to substantial increases in both understanding and run time speed - on single as well as parallel processors.

Modularity & Independence

In engineering, breaking complex systems into independent modules is embodied in the architecture, a concept that has been misunderstood in software. This is because *architecture describes connectivity*, i.e., how a module is connected to other modules. *Engineering architectures represent the time-invariant properties of a system - not flow of control* (they are not flow charts).

Descriptions of architecture are not convenient using algebraic or linguistic representations. Like other engineering fields, software architecture is best described with drawings, depicting how modules are connected. Only then can one visually observe independence - the key property supporting concurrency. Flow charts, or graphical variations on flow charts, are of little use when describing the property of independence.

SOFTWARE ARCHITECTURE

As illustrated in Figure 4, software architects can decompose a system into modules by grouping resources and processes into an *elementary module*. *Hierarchical modules* are created by grouping modules into higher level modules. Figure 4 shows a library module that is sufficiently complex to warrant its own drawing. In general, modules are independent if they share no resources (i.e., they are not connected). Having designed an architecture, developers can implement the data structures and rules using the *resource* and *process* languages. Using this CAD system, resources and processes may be edited directly on the drawing as illustrated in Figure 4. The languages do not permit the declaration of scope rules. It is the architecture that determines how data is shared, and the corresponding independence of modules. Most important, the languages are designed to provide for deep hierarchies in both data structures and rule structures to support the Critical Software Architecture Requirements defined above. Without these language properties, understandability of complex software is difficult.

Parallelism, Architecture, and Decomposition

When striving to take advantage of the inherent parallelism in a system, one must determine the architecture of software modules that maximizes concurrency on a parallel processor. Picking the best set of state vectors is key to solving this problem. Again, best translates to simplicity of transformations and run-time speed.

Having selected Generalized State Space as the framework, the mathematical analogy becomes one of selecting the best set of information vectors (Resources) to represent the system attributes. Depending upon how the resources are designed and structured, the rules (Processes) may be much more simple to understand, build, and modify. This is also determined by the *independence properties of the architecture*, i.e. the interconnection of resources and processes.

Unless one has witnessed directly the development of such architectures, the above discussion may take time to comprehend. Having used it, it is apparent that architecture, as defined here, is as critical to software design as it is to any other engineering discipline, with or without parallel processing. But the ability to design good architectures depends directly on the language. It is why productivity multipliers are very high when using this CAD environment, especially in the support mode when a new person has to understand what another has built. We now turn to the critical importance of language in taking advantage of parallel processors.

Software Spaces & Databases - An Extension Of Mathematics

The CAD software approach described here follows from Shannon's *Mathematical Theory Of Communications*, [10], also known as Information Theory. Based on the binary number system and Boolean algebra, this approach defines a mathematical space wherein the set of characters used to write software is represented by strings of bits or binary numbers. This approach follows from the State Space framework formulated by control theory engineers to simplify complex problems in control system design. State Space extends the mathematics of vectors and matrices, simplifying complex transformations using large vector spaces.

Simplification of complex mathematical problems hinges on selection of a good space, reducing complexity of the transformations and the corresponding time to solve the problem. This is apparent when dealing with multi-dimensional hierarchical spaces as occur in software. Software spaces are determined by the databases used to support transformations (instructions) that implement the application. The entire database represents the overall software space.

If a software application is represented by a continuous-time or discrete-time linear mathematical model, the software and mathematical solutions are essentially the same. However, most software applications require actions based upon events as they unfold, being highly nonlinear. Discrete event simulation provides significant insights into this problem, see [7]. Although time is still the basic coordinate, actions jump to the next scheduled event. The difference is that actions typically depend upon complex decision processes. e.g.,

```
IF A IS TRUE ... SCHEDULE PROCESS_A ... ELSE IF B IS TRUE ... SCHEDULE PROCESS_B
```

Some of the execution steps may involve solving systems of equations. More importantly, they will likely contain statements that SCHEDULE a NEW_EVENT in the future. This facility is necessary in time-based models or real-time systems.

As illustrated in the next section, software is simply an extension of mathematics. The corresponding properties of spaces, and the independence of subspaces and coordinates, apply directly. These properties are critical to designing software architectures containing independent modules that simplify parallel processing while maximizing speed.

The Separation Principle

The underlying principle supporting the visualization of software architectures using engineering drawings is the separation of data from instructions at the language level. Defined in 1982 in the design of the General Simulation System (GSS), this has become known as the *Separation Principle*, [4]. The developers of GSS defined the separate languages used to describe the data structures (*Resources*) and rule structures (*Processes*) illustrated above.

Using the Generalized State Space framework, the Separation Principle is achieved by storing all data in *Resources*. Resources are depicted as ovals in architectural drawings as illustrated in Figure 6. *Processes* containing instructions that implement transformations are depicted as rectangles. The lines connecting them determine which processes have access to what resources. In this figure, each process has a dedicated resource and shared resources. Transformation 1 has state vector A as input, has state vector B for dedicated use, and shares state vector C with transformation 2. Therefore, Transformations 1 and 2 are not *independent*. As used here, the property of *independence* ensures that processes running on a parallel processor produce *complete and consistent* results for a given set of initial conditions.

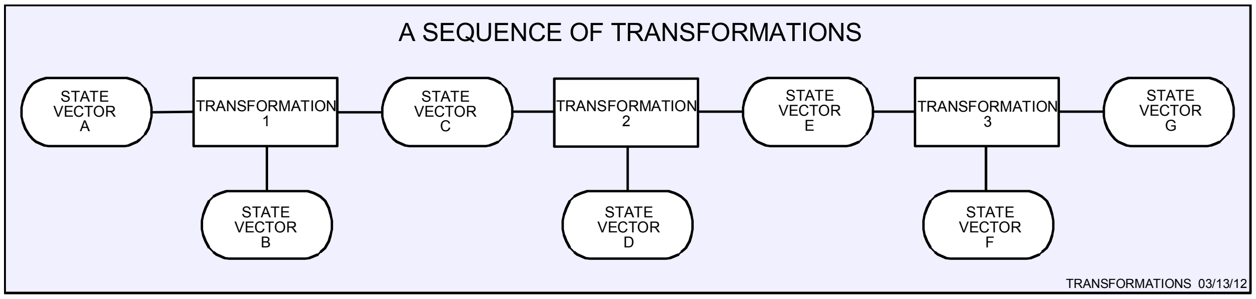


Figure 2. State vectors and transformations.

Consider that state vectors C, D, and E have initial values Ci, Di, and Ei. When run on a single processor (sequential machine), Transformation 2 will produce the same outputs: Co, Do, and Eo for a given set of inputs every time it runs; i.e., the results will be *complete and consistent*. If while it is running, one of the resources is changed from the outside, the results may not be complete and consistent. This is because the data being accessed is not *consistent* relative to Transformation 2. If Transformations 1 and 2 run concurrently, shared state vector C could be changed by either, rendering the data as recognized by the other as *potentially inconsistent*. Therefore, in general, they cannot operate concurrently.

Similarly, Transformation 2 is directly coupled to Transformation 3 by shared state vector E, is not independent of it, and thus cannot run concurrently with it. However, Transformations 1 and 3 can operate concurrently since they share no state vector directly and are therefore *spatially independent*. Transformation 2 can operate only when Transformations 1 and 3 are both idle, i.e., they are *temporally independent.*

The *Separation Principle* provides the ability to represent resources and processes using icons on engineering drawings of software, see Figure 7. Engineering drawings represent the *connectivity* of elements; they are not flow charts. They provide an iconic visualization of which processes share what resources, and therefore their independence. All resources are shared by pointer. By grouping icons into hierarchies of modules, module independence can be visualized directly. Figure 7 is a Library type module.

SOFTWARE LANGUAGE

The requirements for the resource and process languages were driven in part by factors somewhat akin to those motivating the use of *tiling* in parallel versions of FORTRAN. These are to minimize memory management overhead due to swapping processes and paging data. This is accomplished by maximizing the work done on each processor while running concurrently with work on the other processors, thus maximizing the PUE.

To do this, the language must support design of software spaces that simplify the human translation of inherently parallel physical entities into an organization of independent workloads. As understood by Grace Hopper, likely the most knowledgeable software language designer, [3], such organizations are best supported by deep hierarchies of both data and instructions. A simple example of such a data structure (RESOURCE) is shown in Figure 5.

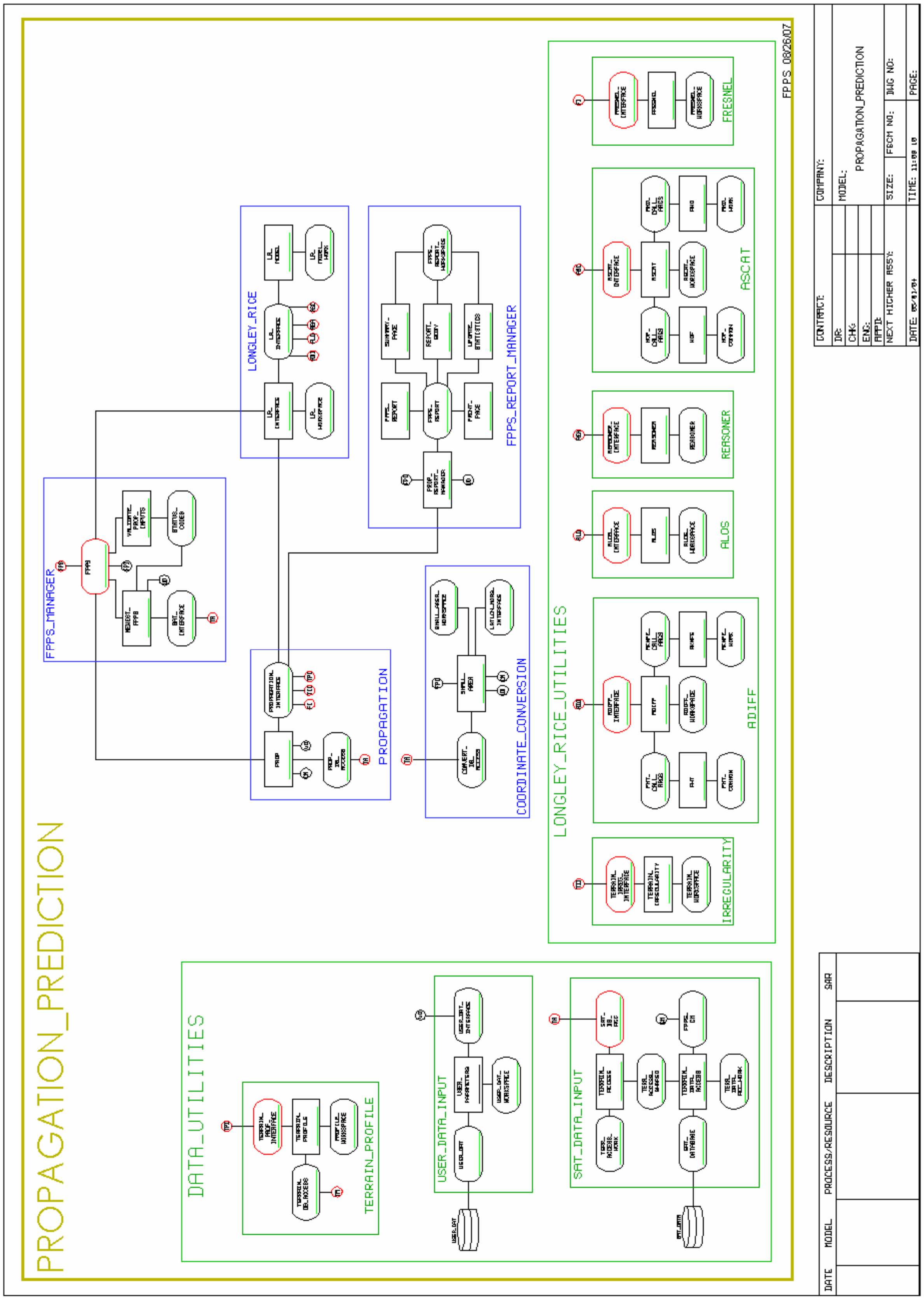
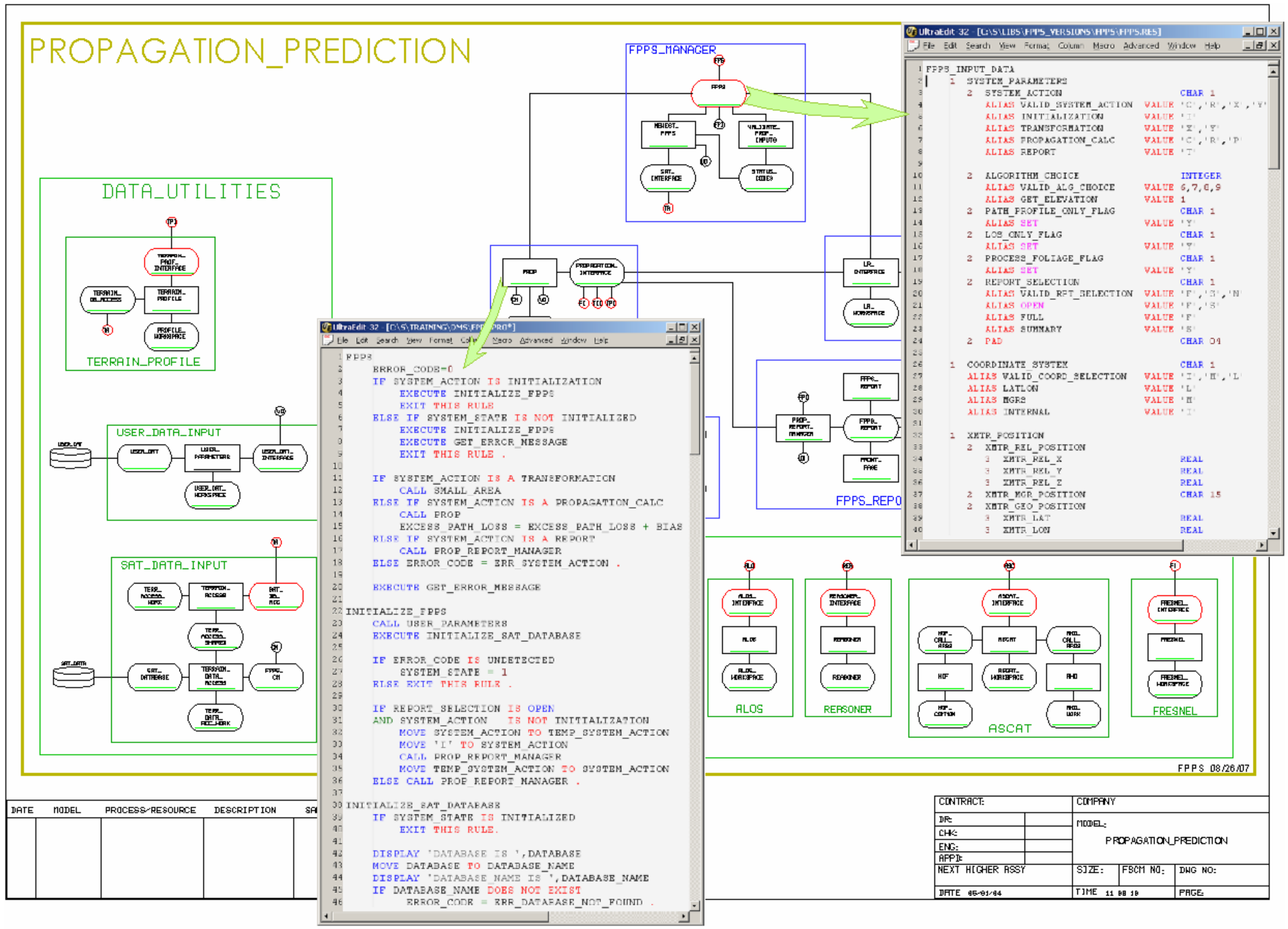


Figure 2. Engineering drawing of a library module.


PROPAGATION_PREDICTION
FPPS_MANAGER
DATA_UTILITIES
TERRAIN_PROFILE
USER_DATA_INPUT
SAT_DATA_INPUT
FPPS_REPO
ALOS
REASONER
ASCAT
FRESNEL
FPPS 08/26/07
UltraEdit-32 - [C:\S\LIBS\FPPS_VERSIONS\FPPS\FPPS.RES]
UltraEdit-32 - [C:\S\TRAINING\DMS\FPPS.PRO*]
File Edit Search View Format Column Macro Advanced Window Help
DATE MODEL PROCESS/RESOURCE DESCRIPTION
CONTRACT: COMPANY
DR: CHK: ENG: APPR:
MODEL: PROPAGATION_PREDICTION
NEXT HIGHER ASSY
SIZE: FSCM NO: DWG NO:
DATE TIME PAGE:

```
FPPS_INPUT_DATA
     1  SYSTEM_PARAMETERS
        2  SYSTEM_ACTION                               CHAR 1
           ALIAS VALID_SYSTEM_ACTION    VALUE 'C','R','X','Y'
           ALIAS INITIALIZATION         VALUE 'I'
           ALIAS TRANSFORMATION         VALUE 'X','Y'
           ALIAS PROPAGATION_CALC       VALUE 'C','R','P'
           ALIAS REPORT                 VALUE 'T'

        2  ALGORITHM_CHOICE                            INTEGER
           ALIAS VALID_ALG_CHOICE       VALUE 6,7,8,9
           ALIAS GET_ELEVATION          VALUE 1
        2  PATH_PROFILE_ONLY_FLAG                      CHAR 1
           ALIAS SET                    VALUE 'Y'
        2  LOS_ONLY_FLAG                               CHAR 1
           ALIAS SET                    VALUE 'Y'
        2  PROCESS_FOLIAGE_FLAG                        CHAR 1
           ALIAS SET                    VALUE 'Y'
        2  REPORT_SELECTION                            CHAR 1
           ALIAS VALID_RPT_SELECTION    VALUE 'F','S','N'
           ALIAS OPEN                   VALUE 'F','S'
           ALIAS FULL                   VALUE 'F'
           ALIAS SUMMARY                VALUE 'S'
        2  PAD                                         CHAR 04

     1  COORDINATE_SYSTEM                              CHAR 1
        ALIAS VALID_COORD_SELECTION     VALUE 'I','M','L'
        ALIAS LATLON                    VALUE 'L'
        ALIAS MGRS                      VALUE 'M'
        ALIAS INTERNAL                  VALUE 'I'

     1  XMTR_POSITION
        2  XMTR_REL_POSITION
           3  XMTR_REL_X                               REAL
           3  XMTR_REL_Y                               REAL
           3  XMTR_REL_Z                               REAL
        2  XMTR_MGR_POSITION                           CHAR 15
        2  XMTR_GEO_POSITION
           3  XMTR_LAT                                 REAL
           3  XMTR_LON                                 REAL
```

```
FPPS
     ERROR_CODE=0
     IF SYSTEM_ACTION IS INITIALIZATION
          EXECUTE INITIALIZE_FPPS
          EXIT THIS RULE
     ELSE IF SYSTEM_STATE IS NOT INITIALIZED
          EXECUTE INITIALIZE_FPPS
          EXECUTE GET_ERROR_MESSAGE
          EXIT THIS RULE .

     IF SYSTEM_ACTION IS A TRANSFORMATION
          CALL SMALL_AREA
     ELSE IF SYSTEM_ACTION IS A PROPAGATION_CALC
          CALL PROP
          EXCESS_PATH_LOSS = EXCESS_PATH_LOSS + BIAS
     ELSE IF SYSTEM_ACTION IS A REPORT
          CALL PROP_REPORT_MANAGER
     ELSE ERROR_CODE = ERR_SYSTEM_ACTION .

     EXECUTE GET_ERROR_MESSAGE

INITIALIZE_FPPS
     CALL USER_PARAMETERS
     EXECUTE INITIALIZE_SAT_DATABASE

     IF ERROR_CODE IS UNDETECTED
          SYSTEM_STATE = 1
     ELSE EXIT THIS RULE .

     IF REPORT_SELECTION IS OPEN
     AND SYSTEM_ACTION    IS NOT INITIALIZATION
          MOVE SYSTEM_ACTION TO TEMP_SYSTEM_ACTION
          MOVE 'I' TO SYSTEM_ACTION
          CALL PROP_REPORT_MANAGER
          MOVE TEMP_SYSTEM_ACTION TO SYSTEM_ACTION
     ELSE CALL PROP_REPORT_MANAGER .

INITIALIZE_SAT_DATABASE
     IF SYSTEM_STATE IS INITIALIZED
          EXIT THIS RULE.

     DISPLAY 'DATABASE IS ',DATABASE
     MOVE DATABASE TO DATABASE_NAME
     DISPLAY 'DATABASE_NAME IS ',DATABASE_NAME
     IF DATABASE_NAME DOES NOT EXIST
          ERROR_CODE = ERR_DATABASE_NOT_FOUND .
```

Figure 3. Illustration of editing resources and processes on the drawing.

```
MESSAGE_TABLE  QUANTITY(3)
    1  MESSAGE_INDEX                           INTEGER
    1  MESSAGE_ELEMENT  QUANTITY(13)
       2  UNIT_I                               INTEGER
       2  SLOT_ID                              INTEGER
       2  MESSAGE_INFORMATION
          3  MESSAGE_TYPE                      STATUS  DATA_OUTPUT
                                                       USER_REQUEST
          3  STATE_S
             4  NUMBER_TO_BE_SENT              INTEGER
             4  SEQUENCE_NUMBER                INTEGER
             4  MESSAGE_ACTION                 STATUS  SEND, HOLD
             4  AGGREGATE_STATE
                5  MESSAGE_STATE  QUANTITY(7)
                                               STATUS  EMPTY, FULL
             4  INDIVIDUAL_STATE  REDEFINES AGGREGATE_STATE
                5  SEQUENCED_MESSAGE
                   6  GROUP_MESSAGE            STATUS  EMPTY, FULL
                   6  BUDDY_MESSAGE            STATUS  EMPTY, FULL
                   6  QUEUED_MESSAGE           STATUS  EMPTY, FULL
                   6  RESERVED_MESSAGE         STATUS  EMPTY, FULL
                   6  INTERCOM_MESSAGE         STATUS  EMPTY, FULL
                5  NON_SEQUENCED_COMMAND
                   6  DATA_INPUT               STATUS  EMPTY, FULL
                   6  USER_COMMAND             STATUS  EMPTY, FULL
```

Figure 5. Example of a hierarchically structured state vector (RESOURCE).

Deep hierarchies allow large complex data structures to be moved in a single instruction fetch, with all of the individual fields directly available to instruction hierarchies as illustrated in Figures 6, 7, and 8. This provides order of magnitude improvements in single processor speeds as well as understanding, see the experimental results in Chapter 17 in [7].

```
PROCESS_CLASS_MESSAGE
    IF   MESSAGE_ACTION(CONTROL_UNIT, RADIO) IS SEND
    AND MESSAGE_TYPE(CONTROL_UNIT, RADIO) IS USER_REQUEST
        MOVE INDIVIDUAL_COMMAND(CONTROL_UNIT, RADIO)
             TO AGGREGATE_STATE(CONTROL_UNIT, RADIO)
        EXECUTE CHECK_MESSAGE_INDEX .

CHECK_MESSAGE_INDEX
    IF MESSAGE_INDEX(TIME_SLOT) IS GREATER THAN ZERO
        SET MESSAGE_ACTION(CONTROL_UNIT, TIME_SLOT) TO HOLD
        SEQUENCE_NUMBER(CONTROL_UNIT, TIME_SLOT) = MESSAGE_INDEX(TIME_SLOT)
        MOVE AGGREGATE_STATE(CONTROL_UNIT, TIME_SLOT)
             TO INDIVIDUAL_COMMAND(CONTROL_UNIT, TIME_SLOT) .
```

Figure 6. Example of part of a hierarchically structured PROCESS.

```
MESSAGE
    1   SYNC_CODE                       CHAR 6
                  ALIAS   VALID       VALUE '101010',
                                            '010101'
    1   TYPE                            STATUS FORMAT_A
                                               FORMAT_B
    1   CONTENT                         CHAR 46

FORMAT_A     REDEFINES MESSAGE
    1   PAD                             CHAR 14
    1   HEADER
        2   PRIORITY                    STATUS FLASH
                                               IMMEDIATE
                                               ROUTINE
        2   ORIGIN                      INDEX
        2   DESTINATION                 INDEX
                ALIAS    BROADCAST              VALUE 0
    1   BODY
        2   LENGTH                      INTEGER
    1   TRAILER
        2   MESSAGE_NUMBER              INTEGER
        2   TIME_SENT                   REAL
        2   TIME_RECEIVED               REAL
        2   ACKNOWLEDGEMENT             STATUS RECEIVED
                                               NOT_RECEIVED
        2   LAST_SYMBOL                 CHAR 2
              ALIAS   TERMINATOR      VALUE '\\', '//', '<<','>>'

FORMAT_B     REDEFINES MESSAGE
    1   PAD                             CHAR 14
    1   HEADER
        2   SOURCE                      INDEX
        2   SINK                        INDEX
    1   BODY
        2   CONTENTS                    CHAR 42
```

Figure 7. Example of a hierarchically structured state vector (Resource).

When building complex software, human translation is simplified if a language supports obvious representation of physical behavior. The examples in Figures 7 and 8 are taken directly from large detailed simulations of Packet Radio networks. With hierarchical data structures like those shown, one can represent the complex algorithms associated with physical systems with ease. This is illustrated in the above figures. Actual systems may entail more complex resources and processes than those shown, but are easily understood by subject area experts.

Not shown in Figure 7 are the QUANTITY clauses used in Figure 5. Likewise, similar corresponding subscripts in Figure 5 are not used in Figure 8. This is because the resource and process pair are part of an instanced module, where instances are automatically handled at the module level, being set when a process within an instanced module is CALLed or SCHEDULEd. Moving instance implementation to the module level substantially enhances understanding of the code.

```
 PROCESS: RECEPTION

 RESOURCES: TERMINAL_PARAMETERS      INSTANCES: TRANSMITTER
            MESSAGE_FORMATS                     RECEIVER
            TRANSCEIVER

START_RECEPTION
    IF TRANSCEIVER IS IDLE
        EXECUTE GOOD_RECEPTION
    ELSE IF TRANSCEIVER IS RECEIVING
        EXECUTE CONFLICTING_RECEPTION
    ELSE IF TRANSCEIVER IS TRANSMITTING
        EXECUTE CONFLICTING_BROADCAST .

GOOD_RECEPTION
    IF SIGNAL_TO_NOISE_RATIO IS GREATER THAN

RECEIVER_THRESHOLD
         SET TRANSCEIVER TO RECEIVING
         ADD SIGNAL POWER TO POWER_AT_RECEIVER
         CALL DECODE_MESSAGE
    ELSE EXIT THIS RULE .

    IF SYNC_CODE IS VALID
    AND LAST_SYMBOL IS A TERMINATOR
    AND MESSAGE TYPE IS FORMAT_A
        EXECUTE SEND_ACKNOWLEDGEMENT .

CONFLICTING_RECEPTION
    IF POWER_AT_RECEIVER IS GREATER THAN SIGNAL_POWER
         SCHEDULE ABORT_RECEIVE NOW .

CONFLICTING_BROADCAST
    CANCEL END_RECEIVE NOW
    SCHEDULE START_RECEIVE IN EXPON(0.83) MILLISECONDS
        WITH PRIORITY 80

SEND_ACKNOWLEDGEMENT
    MOVE ACKNOWLEDGEMENT TO TRANSMIT_MESSAGE_BUFFER
    IF DESTINATION IS BROADCAST
         SEARCH RECEIVER_CONNECTIVITY_VECTOR OVER RECEIVER
              EXECUTING TRANSMISSION
                   WHEN LINK IS GOOD
    ELSE EXECUTE TRANSMISSION .

TRANSMISSION
    SCHEDULE RECEPTION
       IN LINK_DELAY MICROSECONDS
           USING TRANSMITTER, RECEIVER
```

Figure 8. Example of a hierarchically structured transformation (Process).

Selection of the proper type of resource is a critical architectural decision when designing complex software systems. Resource types determine the simplicity of the architecture, invoking substantial VisiSoft facilities that are built into the environment. From the Visual Development Environment (VDE), designers can create or modify resources using the corresponding buttons and panels. The panels provide the ability to explicitly specify the types of resources desired and enter the corresponding information required for a given type. Each resource must be explicitly defined as one of the following sharing types.

Table 1. VisiSoft Resource Types.

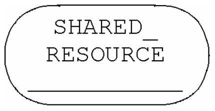
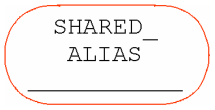
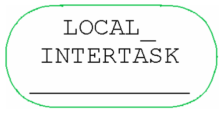
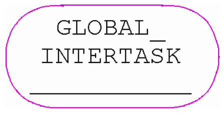
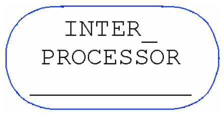
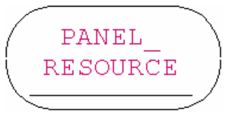
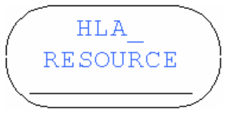
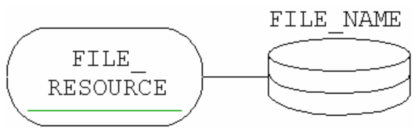
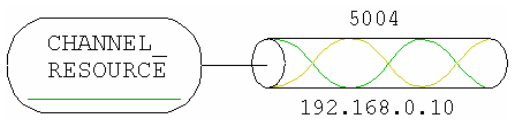
	Resources hold state data that may be organized in a hierarchical manner. It can be shared by several processes or dedicated to a single process in a single task. Connection to either a file or communications channel makes the resource “dedicated” to a single process.
	A resource with a memory template, typically for a utility or library module. Shared Alias resources are outlined in red. They are provided a pointer to the actual resource. See examples below for more details.
	A Local Inter-Task resource allows a family of tasks to share data. VisiSoft handles the OS level memory management. Local Inter-Task resources are used when one task is responsible for “STARTing” another one that shares the same local Inter-Task resource. Local Inter-Task resources are outlined in green. See examples below for more details.
	Global Inter-Task resources are similar to Local Inter-Task resources. A Global Inter-Task resource is used to allow two tasks to share data when they are RUN *independently* rather than when one task STARTs the other. Global Inter-Task resources are used for SYSTEM level EVENTS. They are outlined in purple. See VSE examples below for more details.
	An Inter-Processor resource is used to share data between IND modules on different processors in the same task running on a parallel processor. Inter-Processor resources are outlined in blue. See examples below for more details.
	PANEL resources are used to support graphical panel interfaces for input and output of information, which can include icons, scrolling lists, etc. The contents of a PANEL resource are created and modified using the Panel Library Manager (PLM) - see the PLM Section of the RTG Manual. The contents of a PANEL resource may be viewed, but not changed with a text editor. Red text is used to label a PANEL resource.
	An HLA resource supports the use of High Level Architecture for communications between disparate tasks in a multi-task environment. This resource and an associated HLA event handler enable easy use of HLA from within a VSE task. Details on the use of this resource type are described in Section 15 on HLA Interface. An HLA resource is labeled with blue text.
	A resource describing the record(s) on the file to which it is attached. The FILE_NAME identifies the name of the actual file to be accessed.
	A resource describing the TCP/IP channel to which it is attached. The number on top of the channel icon is the PORT number and that underneath is the SERVER ID.

Table 1 above provides another illustration of the facilities required for engineering the architectures of complex software systems.

VisiSoft Module Types

There are four types of modules that make up the layers of a software design hierarchy. These types provide different levels of protection with regard to their reuse in different hierarchies. Both elementary and hierarchical modules can reside within each type. With the exception of instanced utilities, modules may only appear once in a drawing. The rules for these types are described below with examples that follow.

- **Modules** - have a blue border. These are the basic building blocks in a task. In the CAD system described here, modules may be decomposed hierarchically, i.e., they may contain submodules and sub-submodules, etc. Modules may only appear in a single drawing in a user directory, and are meant to be unique, i.e., not reused, across directories.

- **IND Modules** - have a blue border. IND Modules are Modules that can only share Inter-Processor (IP) Resources - and only with other IND_Modules. When using parallel processors, IND_Modules must be the highest level modules on a processor. IND_Modules may reside on the same or different processors.

- **Utility Modules** - have a green border. These are modules that are reused by processes in the same directory, and can appear in more than one hierarchy in different drawings. They are typically used to manage separate databases or perform utility type functions. The green color distinguishes them for change protection. If they are changed to accommodate a different requirement, that change must be compatible with the other processes that use them, since the change is automatically embodied in them all. A separate copy resides on each processor that uses it.

- **Library Modules** - have a gold border. These are more highly protected utility modules that can be shared from different directories and different computers. They are stored as object modules in special object library files. The source only appears in the directory where they are maintained. Processes in a library module are called from an application using their process name, module name, and library name. Since each of these names must be unique within the next level of hierarchy, there can be no duplicate names when linking to library modules in the CAD environment described here. A separate copy resides on each processor that uses it.

  The functions of a library module may be upgraded while at the same time preserving the original module in the library for prior users. Users can call the new function using the same process name within the same library by using the new module name. The existing CAD system has a large set of libraries that support various applications, including 3D graphics, that are shared easily.

  The CAD libraries have been designed to be controlled separately under special protection mechanisms. But given access to a library directory, the responsible person sees everything that is needed to allow for ease of changes and testing. Library directories typically contain regression test drivers and data sets to ensure changes meet all prior, as well as new requirements.

INITIATING AND TERMINATING VSE TASKS

Implementation of operating systems and real-time control systems require multiple tasks. VisiSoft Tasks can be initiated and terminated in different ways. The approach depends upon the type of task desired. Global or "top level" tasks are initiated using the RUN statement.

GLOBAL VSE TASKS

The RUN statement can be used to RUN a task directly, or to initiate a script that RUNs a task. In either case, a Global VSE Task is initiated. In Figure 11 below, TASK_1 is a Global Task that RUNs TASK_2, another Global Task. TASK_2 then RUNs TASK_3 which becomes a third Global Task.

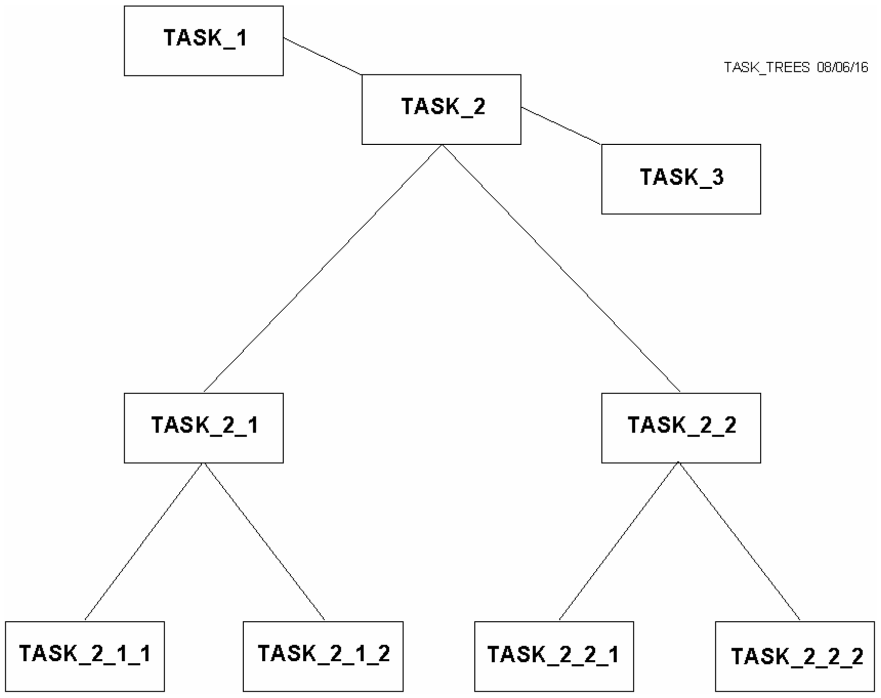


Figure 11. An example of VSE TASK TREES.

Once a Global Task is running, it may initiate additional tasks. These additional tasks may be Global or Local. Global VSE tasks may share GLOBAL Intertask resources with other GLOBAL tasks. Global tasks may only be terminated by themselves.

LOCAL VSE TASKS

Any VSE task may use the START statement to start a LOCAL task. Tasks that are STARTed by another VSE task are LOCAL to the task that STARTs them, and are considered part of the STARTing Task's family. In Figure 11, TASK_2 starts Local tasks TASK_2_1 and TASK_2_2. They become part of TASK_2's family's tree. Each of these tasks starts the two below it. TASK_2_1_1 is part of TASK_2_1's family tree. TASK_2_2_2 is not, but belongs to TASK_2_2's family tree as well as that of TASK_2.

Local VSE tasks may be terminated by themselves. They may also be terminated by a task that is a higher member of the tree to which they belong. They are also terminated automatically when a task that is a higher member of the tree to which they belong is terminated.

USE OF SHARED AS - ALIASED RESOURCES

When a utility is CALLed directly from more than one process, the calling processes typically share a resource whose attribute structure is common, or ALIASed, with the utility.

Referring to Figure 3-7, SHARED AS resource UTR_INT is ALIASed as resources UMR_1 and UMR_2. The resource structure template used by process UTP is UTR_INT. When UMP_1 or UMP_2 calls UTP, UTP will use either resource UMR_1 or UMR_2 respectively. The attributes used in UMR_1 or UMR_2 can be different, as long as the data structure they represent maps into the template defined by UTR_INT. The template defined by UTR_INT is available to as many calling processes of UTP as desired, without any need to modify UTP. To accomplish this, the designer must click SHARED ALIAS'D in the Sharing type section of the panel while creating the resource in the CALLed utility. For example, UTR_INT would then be defined as SHARED ALIAS, and its outline would be colored red.

Any resource connected to a SHARED ALIAS resource that resides in the directory (library modules may not) automatically becomes a SHARED AS resource to the ALIAS resource identified in the called process to be used. For example, UMP_1 uses UMR_1 SHARED AS UTR_INT when calling UTP. Up to eight alias resources can be connected to each SHARED AS resource in a calling process.

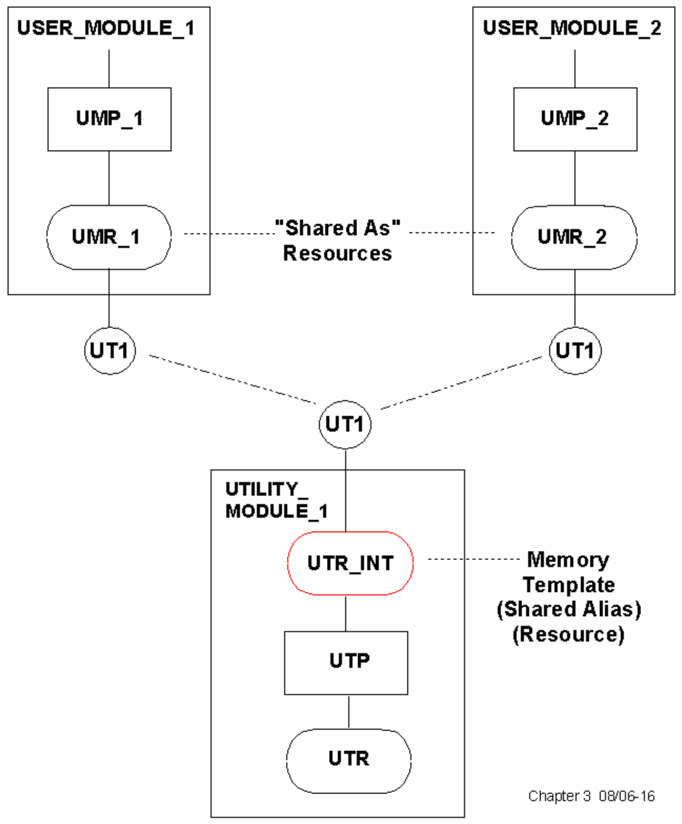


Figure 9. Use of an ALIASed Resource.

USE OF INTERTASK RESOURCES

Intertask resources afford communications between multiple tasks, as supported by a multitasking operating system. VSE provides special facilities that eliminate the need for designers to deal with the operating system and special shared memory calls. An intertask resource is used the same way as a normal shared resource, except that it is shared across tasks. In Figure 3-8, processes that share intertask resources can access the attributes of those resources just as normal resources. When concurrent tasks share an intertask resource, it is up to the designer to use the facilities to insure data coherency, i.e., that data is not updated incorrectly, e.g., when one task writes over what another expects to be unchanged.

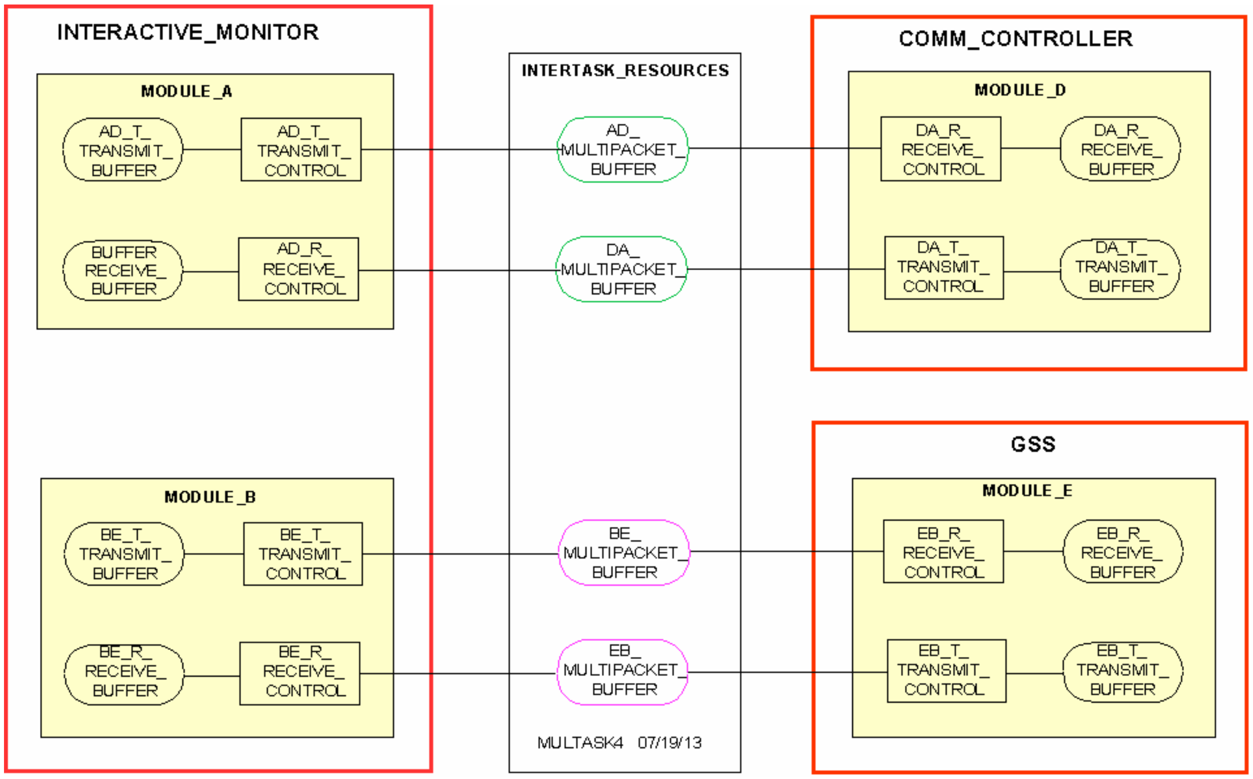


Figure 10. Use of INTERTASK resources.

The difference between LOCAL and GLOBAL intertask resources is illustrated by the colors in Figure 3-8. By virtue of its LOCAL intertask resources, COMM_CONTROLLER is part of a task family with the INTERACTIVE_MONITOR. The GSS task may or may not be part of the family. It may still share global intertask resources.

When TASK_A interfaces with TASK_B, and TASK_B may be sharing intertask resources with other tasks, (e.g., RTG), then the names of these resources must be known for TASK_B to be part of a TASK_A family. If members of an intertask family share different intertask resources that happen to have the same name, they will be considered the same - as part of the family. A case in point is when a pair of tasks both use RTG. These tasks must not be part of the same family.

USE OF INTERPROCESSOR RESOURCES

To maximize speed of full duplex communications, one must limit Inter-Processor (IP) resource connections to One-To-One and One-To-Many, as shown in Figure 3-9. With this approach, only processes inside the IND module containing an IP Resource may write to it. This ensures that only one process can write to an IP resource at a time. This is the easiest approach for an application expert to use to create the best (fastest) architecture.

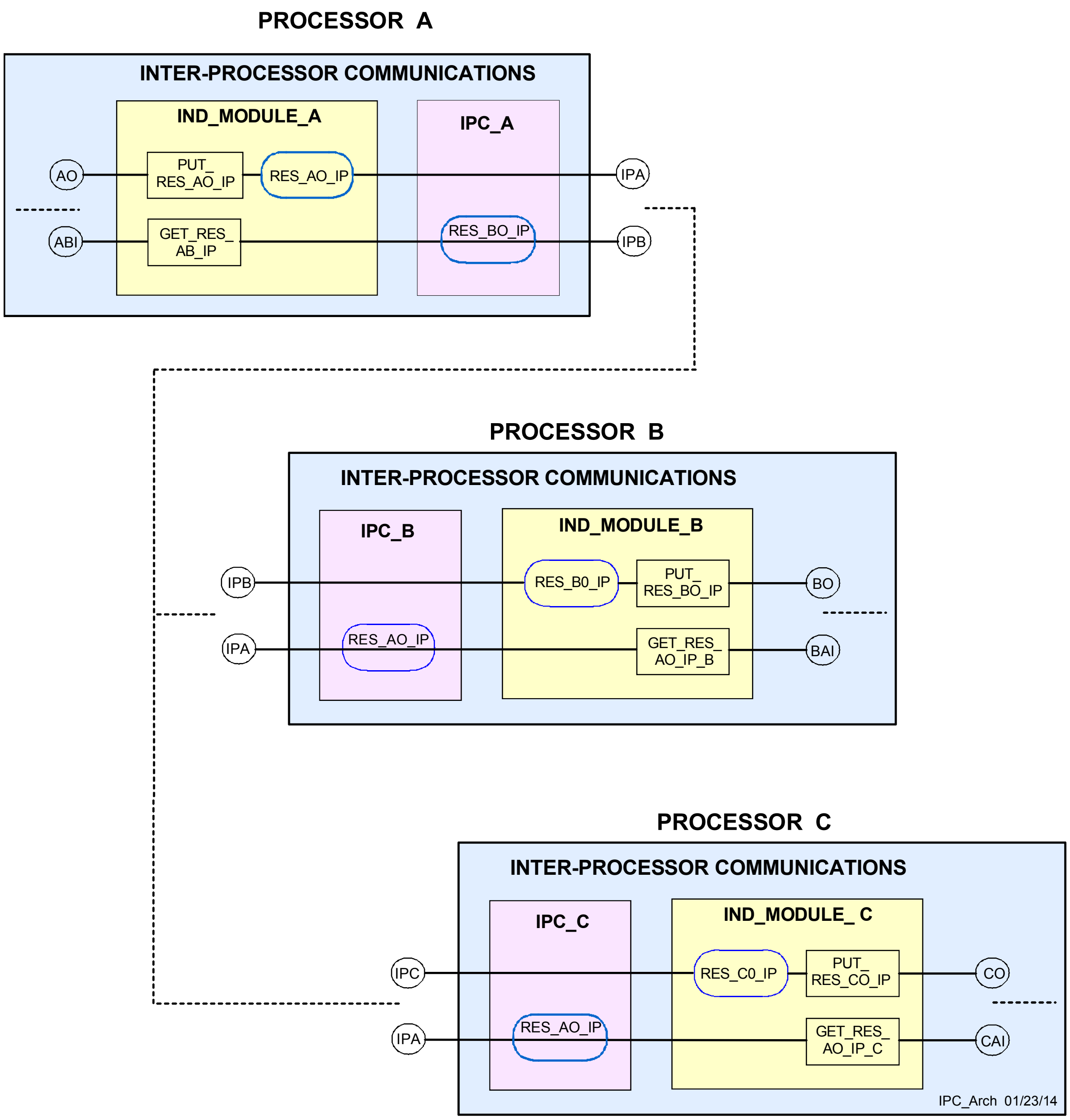


Figure 3-9. IP Communications (IPC) Architecture contained in the Run-Time System.

Considering the most difficult application that PSI has faced - determination of Electro-Magnetic Wave propagation, this approach does not impose any restriction on architects that are knowledgeable in the application they are building. It does help them to develop and enforce good architectures that maximize run-time speed.

When using VSE Inter-Task Resources, the resource is shared directly between tasks. However, Inter-Processor (IP) Resources are effectively copied between processors automatically by the IP Communications (IPC) system. Copies must be RELEASEd by the module that contains the IP resource, and ACCESSed by the module wanting to read it. Only processes in the same IND module can write to an IP resource, and copies of IP Resources are maintained within each IND module that reads the IP resource. When a process that writes to an IP Resource completes, IPC code at the end of that process updates a system resource in the IPC module, indicating that a modification has been made.

Resource coherency of IP Resources is implicit because memory is moved using a single instruction, blocking out other instructions while it is performed. This implicitly ensures resource coherency with no overhead. Both the reading and writing processes may run concurrently. By making copies, memory is used to gain speed. All of this simplifies the IPC architecture in the VPOS Run-Time System.

Architecture Of An Instanced IP Resource Facility

When using parallel processors, particularly in a simulation, one often uses instanced IND modules. This is extremely convenient since instance pointers are handled implicitly within an instance that is scheduled or called by a process that specifies the instance to be used while memory is copied explicitly. Automatic instancing of the corresponding IP Resources provides the same level of simplification to the user. Equally important it provides a significant simplification of the architecture for IP Resource communication.

SUMMARY

Design of complex automation systems that require advanced computer technology represents a difficult engineering problem. This paper briefly skims aspects of system design to illustrate some of the top level concepts. It is intended to demonstrate an architectural perspective that equates to similar engineering fields, e.g., aeronautical, architectural, electrical, etc. In those fields, the people that implement the construction of an end product are not the designers. Carpenters, electricians, masons, and plumbers require years of training to achieve the skills necessary to perform their specific tasks. But their skill sets can be learned on much smaller projects. The engineering skills required to design skyscrapers requires years of working with large complex structures. They cannot be learned on dog houses. Learning to program snippets of code using different languages is akin to learning the trades.

It has been the purpose of this paper to show that complex automation problems cannot be solved with programming languages. The language must support architectural requirements, but it is the architectural technology that is required to design large complex systems and simulations, particularly on parallel processors. This paper has been aimed at disclosing this pressing requirement to refocus the software field toward an engineering discipline.

REFERENCES

[1] Ledgard, Henry F., *The Emperor with No Clothes*, Communications of the ACM, Oct 2000.

[2] Parnas, D., "Education for Computer Professionals," IEEE Computer, January 1990, pp 17-22.

[3] Beyer, Kurt W., *Grace Hopper and the Invention of the Information Age*. Cambridge, MA: The MIT Press, (2009). ISBN 978-0-262-01310-9.

[4] Yasushi Kambayashi and Henry F. Ledgard, "The Separation Principle - A Programming Paradigm" IEEE Software, March/April 2004

[5] Broy, Manfred, *The "Grand Challenge" in Informatics: Engineering Software-Intensive Systems*, Computer, IEEE Computer Society, 2006.

[6] Poore, Jesse H., *A Tale of Three Disciplines and a Revolution*, IEEE Computer Society, Jan 2004.

[7] Cave, W.C., et.al, **Software Theory For Parallel Processors**, Visual Software International Technical Report, March 2016, Spring Lake, NJ.

[8] Deming, W. Edwards, *Out of the Crisis,* MIT CASE, Cambridge, MA, 1992.

[9] Ledgard, H., et al, "The Natural Language of Interactive Systems," CACM No. 10, October 1980, pp 556-563.

[10] Shannon, C.E., A Mathematical Theory Of Communication, BSTJ, Vol.27, pp 379 & 623, Jul & Oct 1948.