Theory And Design Of Concurrent Processing Facilities

† W.C. Cave, R.E. Wassmer, K.T. Irvine - 06/06/17

Purpose of the Theory - Simplify development, support and operation of application systems that require parallel processing, e.g., modeling / simulation and real-time control systems.

Measurable Goals

To fairly compare solution approaches, one must observe a wide spectrum of applications that are critical to the future of U.S. technology and are known to be difficult to address effectively. In the case of simulation, application experts must understand the models as implemented in the software to ensure that a system is correctly represented. One must then focus on fair measurement of results relative to achieving application needs. Only application experts can judge whether those needs are met. This implies the following measurable goals:

- Provide multiple orders of magnitude improvement in application run-time speed;
- Provide an order of magnitude reduction in the time and cost to develop software;
- Allow application experts to design, build, and test software directly;
- Allow newcomers to a project to quickly learn and understand complex software.

Representative Applications

Those who have worked many parallel processor applications know that each class of applications appears different. Yet many share common properties. The applications listed below represent an immediate market, and can be used to characterize common properties that directly affect the measurable goals stated here.

1. Real-Time Control of Large Groups of Autonomous Moving Platforms
2. Human Body Organ simulation
3. Global Climate prediction
4. Fluid Flow simulation
5. Biological Particle simulation
6. Chemical - Molecular structure simulation
7. Scanning, sorting, and correlating massive databases (Big Data)
8. Weather prediction in mountainous terrain
9. Power distribution simulation
10. Electro-magnetic wave simulation
11. Global HF power transmission
12. Global Military Planning - Multiple moving platform simulation

† Authors are with Visual Software International

Operational Systems Must Meet Application Requirements:

- Operational accuracy
- Speed constraints.

Using A Single Processor To Implement The Required Decision Processes

When using single processors to implement decision processes to achieve the desired application outcomes, one typically uses some form of differential representation of evolving actions as a function of time. On a single processor, this must be implemented using a set of sequential instructions. This implies selecting a ΔT that is sufficiently small so that the result of evolving outcomes is represented with sufficient accuracy at the end of each ΔT increment. In the case of large complex systems, these decision processes may take substantial amounts of time. If the application time constraints cannot be met with a single processor, then one may use a parallel processor. However, the development of data spaces and corresponding algorithms will likely be substantially different to take advantage of the potential parallel processing speeds.

In the case of nonlinear or nonstationary systems, looping through repetitive sets of calculations and decisions may be required to ensure convergence of solutions to meet predetermined or evolving error criteria. In the case of large complex systems, these repetitive processes may take substantial amounts of time. Again, one may meet the required run-time constraints using parallel processing, and system development will likely be substantially different to achieve the potential parallel processing speeds.

Using Parallel Processors To Implement The Required Decision Processes

The obvious difference when using parallel (as opposed to single) processor systems is the ability to perform instructions concurrently on separate processors. The obvious implication is that the application has inherent parallelism in it. If not, then those instructions must be sequential. Clearly applications that can be sped up using parallel processors have substantial inherent parallelism.

One must beware of measures that are based on "Embarrassingly Parallel" applications. Embarrassingly parallel applications can be split easily into totally independent parts that may run concurrently with little if any intra-communications, except at the start and end of a run.

For example, in the case of parametric analyses, one may run many simulations using different random number seeds to observe how parameter variations that fall randomly within a distribution affect the outcome of a sequence of events. These types of problems may be run on servers or a cluster of small machines (e.g., PCs) since the individual simulations are virtually independent (they share no data while running). This property underlies the definition of problems known as "Embarrassingly Parallel". It implies they may be designed to run independently - in serial or parallel - to obtain the same answers. When one is concerned about overall run time, the set of sequential tasks may be run - independently - in parallel.

None of the representative applications listed above are embarrassingly parallel. These applications generally require a totally different approach to their software (and hardware) design, one that maximizes Processor Utilization Efficiency (PUE), see [6].

DESIGNING COMPLEX SYSTEMS

Development of complex systems that do not need the speed of parallel processors has been a major problem in the software field. It has been well described in the literature concerned with the development of large systems since the 1980s, see [1], [2], [3], [7], [9], [11], [12], [13], [14]. When comparing development of these systems to that of large architectural structures, e.g., buildings, aircraft, power plants, communication networks, etc., these software application systems should be easier to build. So what is the problem?

Engineering Drawings

If one looks at the current methods for building software, one finds critical faults in the approach. These faults are similar to those observed in the early centuries as engineering fields evolved. For example, architectural engineering did not exist until more recent times. Artists provided renditions of what a building should look like. There were no engineering drawings and no measurements. The art was turned over to builders who worked to figure out how to do the construction, often redoing their work during development. Today, skyscrapers require hundreds of engineering drawings and books of specifications to ensure that construction proceeds in a well defined, orderly and high quality manner. Masons, carpenters, plumbers, electricians, etc. are expected to follow the drawings and specifications. They do not do the designs as they go. Architects inspect their work as it progresses.

Architecture

Although the word architecture is well defined when used in hardware engineering disciplines, it remains unspecified when used in software literature. Similarly, the word module is well defined in hardware engineering fields. Its use in software appears to imply similarity to hardware but remains undefined. The word module is well understood in computer chip and board design, but clearly misunderstood when used in relation to software. One does not build complex hardware systems without first creating an architecture. The architecture is defined using engineering drawings.

Modules

Complex hardware systems define modules using engineering drawings. Their use is critical when it comes to maintenance. Modules must be easy to replace when they fail. Therefore, they are designed to be independent, implying they can be pulled and replaced easily when they fail. The property of independence also provides the ability to change the interior parts of a module without affecting the use of its external connections or use by other modules.

Architecture Is Time Invariant

Architecture is time invariant. It does not describe flow of control, e.g., a flow chart. It describes the connectivity of components that make up the modules, and the interconnection of components in one module to those in another.

AN ENGINEERING APPROACH TO BUILDING COMPLEX SOFTWARE

The answer to the general problem of expanding technology is to linearize growing complexity. Engineers, scientists, and mathematicians study linear systems in school. They are taught how the property of independence can be used to linearize a complex nonlinear system. This is accomplished by using spaces that fit the problem, so that each coordinate is independent of the others. More generally, mapping the application into a set of independent parts can effectively linearize an otherwise nonlinear problem, making solution approaches much more understandable.

When working as a team in the development of complex systems, it becomes apparent that complexity is reduced by breaking such systems into a hierarchy of spaces. The selection of spaces determines the level of simplification of the design. This concept is no different from the solution of complex mathematical problems where the selection of the "best" space leads to maximum simplicity of the algorithms. This occurs when the subspaces are maximally independent. This results in simplification of the algorithms and reduction of the time to achieve solutions. Selecting the best space requires deep understanding of the dynamics of the problem.

The applications listed above require designing the best spaces in which to map the inherent parallelism in each into software. As shown in Figure 1, this requires mapping the application requirements into an overall software/hardware space that is defined by experts using a software environment that aids in the design effort. This environment must support ease of translation of the application requirements into a software space that maximizes understanding as well as speed. This is accomplished in various engineering fields using Computer-Aided Design (CAD) systems. Then the very complex translation into a hardware language is shifted to the computer - just where the burden should be.

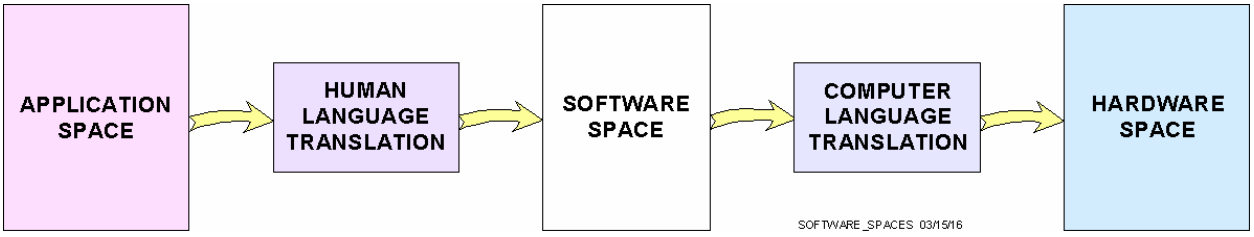


Figure 1. Spaces for translation of application requirements into software and hardware.

Instruction Set Architecture (ISA)

When Von Neumann defined requirements for the first stored program digital computer in the early 1950s, he based it on a variety of applications, from weather prediction, to solving ballistic equations, and solving the compression equations to produce enough energy to set off a nuclear reaction. He derived the *Instruction Set Architecture* (known as the ISA), the set of instructions needed to write the computer programs to implement a wide range of applications. While he did not design the first stored program digital computer (the MANIAC), he provided the programming interface requirements (the ISA) to those who did the logical design to implement his stored language instructions. Memory size quickly became the key to speed.

Application Space Architecture (ASA)

The representative applications enumerated above all have different software design requirements. As in most difficult applications, the mathematical spaces selected to minimize the complexity of their solutions are all different. Representing these problems requires hierarchical data spaces and discrete event spaces - as well as continuous and discrete time spaces - all in the same application. To support this translation requires an extension of mathematics that helps one to conceive the design of these spaces and corresponding decision processes. The application software must also invoke synchronization facilities for sharing temporally independent data spaces (independent at specific times). Only application experts can fully understand these spaces and their independence properties. Such software development facilities must be easy for these experts to use. Given languages that efficiently implement these facilities, one is on the road to determining the requirements for parallel processor hardware design.

Simplifying the software design process for parallel processing requires more than just new programming languages or hardware facilities. It requires the ability to map the inherent parallelism of an application system into software modules that can run concurrently. This is met when application experts can easily visualize the architecture of the application system and the corresponding software design. The key properties required of application software architecture are the *understandability* and *independence* of modules. These requirements are met in engineering fields where CAD systems (e.g., for electronic circuit design) use visualization to create architectures that represent higher levels of the design process that are most critical.

Design of software for an HPC application requires decomposition into a hierarchy of independent modules that may run concurrently. Given a sufficient amount of inherent parallelism, one must map that parallelism into a *software architecture* tailored to the specific application. To develop optimal decompositions, one must start with the top level requirements, working down to the bottom of the algorithms. This process requires application expertise.

What is *software architecture*? It is the same as other engineering architectures. Skyscrapers, airplanes, ships, electronic circuits and chips could not be produced without the visualization of engineering drawings. Engineering drawings define many critical system properties. First, they are time independent - they do not represent flow of control. Second, they provide a visualization of the connectivity (independence) of modules designed to implement the system. Independence is critical to both construction and maintenance of complex systems. Visual depiction of the independence of modules is critical to their design and their ability to run concurrently, i.e., in parallel.

Visual Software Engineering

To meet parallel processor speed constraints, software architectures must be decomposed into independent modules that can share data while running concurrently. To accomplish this, application experts must be able to easily understand the detailed design - down to the code. These requirements have led to a visual software engineering approach, using a CAD system for designing parallel processor software described in [6], and illustrated in Figure 2.

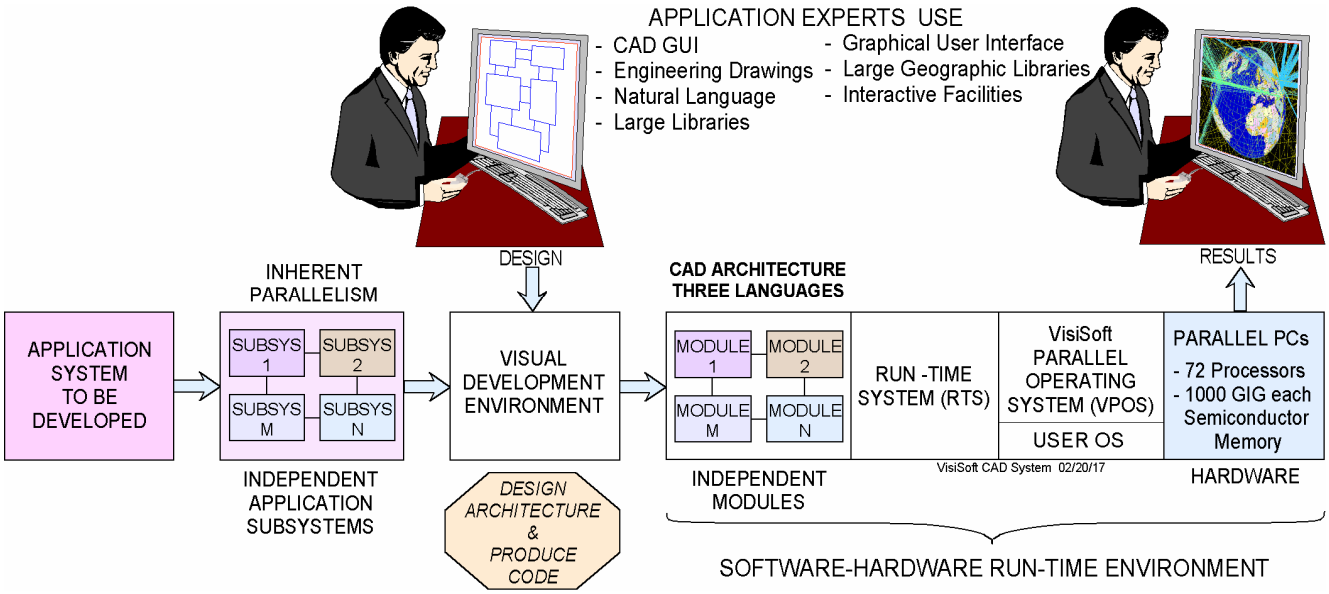


Figure 2. Visual software engineering using a CAD system.

The context of architecture as used here is no different from the hierarchy of drawings required to support the design of skyscrapers. This technology is unnecessary when building dog houses or small software applications. Similarly, architectural drawings must be accompanied by a set of easily understood specifications. This raises the question: What does it take to produce engineering drawings of software, and languages that are easily understood by application experts?

VisiSoft Languages

The VisiSoft languages, [5], have been designed to ease the understanding of detailed algorithms required to implement extremely complex systems. As understood by Grace Hopper, such organizations are best supported by deep hierarchies of both data and instructions using an English-like language. This is particularly true when dealing with large modules that must be designed by subject area experts to run independently.

The Separation Principle

The underlying principle supporting the characterization of module independence is the ability to determine which modules share what data. This is accomplished by separating data from instructions at the language level. Defined in 1982 by Cave while designing the General Simulation System (GSS), this has become known as the *Separation Principle*, [8]. GSS has evolved into what is now VisiSoft, a CAD system for software, [5]. Using what is defined as the Generalized State Space framework, the Separation Principle is achieved by storing all data in *Resources*. A relatively simple Resource is illustrated in Figure 3.

```
RESOURCE: TRANSCEIVER                    INSTANCES: TRANSMITTER
                                                    RECEIVER

GENERAL_PARAMETERS
    1  TRANSMITTER_POWER            REAL   INITIAL_VALUE 100
    1  RECEIVER_THRESHOLD           REAL   INITIAL_VALUE 120

RADIO
    1  TRANSCEIVER                  STATUS TRANSMITTING
                                           RECEIVING
                                           IDLE
                                           OFF
    1  LOCATION
       2  X_POSITION               REAL
       2  Y_POSITION               REAL
       2  ELEVATION                REAL
    1  ANTENNA_HEIGHT               REAL
    1  ANTENNA_GAIN                 REAL

RECEIVER_CONNECTIVITY_VECTOR
    1  POWER_AT_RECEIVER            REAL
    1  TOTAL_NOISE_POWER            REAL
    1  CONNECTIVITY_MATRIX
       2  PROPAGATION_LOSSES
          3  TERRAIN_LOSS           REAL
          3  FOLIAGE_LOSS           REAL
          3  TOTAL_LOSS             REAL
       2  SIGNAL_POWER              REAL
       2  SIGNAL_TO_NOISE_RATIO     REAL
       2  LINK_DELAY                REAL
       2  LINK                      STATUS GOOD
                                           FAIR
                                           POOR

TRANSCEIVER_RULES
    1  TRANSCEIVER_PROCESS          RULES  GOOD_RECEPTION
                                           CONFLICTING_RECEPTION
                                           CONFLICTING_BROADCAST
```

Figure 3. Example of a hierarchically structured state vector (RESOURCE).

Deep hierarchies are required to design complex spaces. They must also allow large complex data structures to be moved in a single instruction fetch, with all of the individual fields directly available to instruction hierarchies in *Processes* as illustrated in Figure 4. This provides ease of understanding as well as orders of magnitude improvement in single processor speeds. Experimental results of speed comparisons are described in [6], Chapter 18.

Note that subscripts are not used in Figures 3 or 4. This is because the resource and process shown are part of an instanced module, where instance pointers (TRANSMITTER & RECEIVER) are automatically handled at the module level. These are 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: TRANSCEIVER
INSTANCES: TRANSMITTER                           MESSAGE_FORMATS
           RECEIVER                              TRANSMITTER_OUTPUT

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 TOTAL POWER_AT_RECEIVER .
        CALL DECODE_MESSAGE .

    IF  MESSAGE TYPE IS FORMAT_A
    AND SYNC_CODE    IS VALID
    AND LAST_SYMBOL  IS A TERMINATOR
         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 LINK_CONNECTIVITY_VECTOR OVER RECEIVER
             EXECUTING TRANSMISSION
                  WHEN LINK IS GOOD
    ELSE EXECUTE TRANSMISSION .

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

Figure 4. Example of a hierarchically structured transformation (Process).

More importantly, VisiSoft allows users to assign selective sets of instanced modules to specified processors using the third language, the Control Specification. The assignment process is aided by a graphical interface so applications experts can take advantage of their knowledge of the physical properties of the application relative to physical processor assignments. This language provides many facilities, e.g., sections for defining files, libraries, initialization and multiple reruns for simulation and optimization, as well as eliminating scripts.

Most important, the language supports design of software spaces that simplify human translation of inherently parallel physical entities into organizations of independent workloads or modules. Design of the resource and process languages were driven in part by factors somewhat akin to those motivating the use of *tiling* in parallel versions of MPI Fortran. These are to minimize memory management overhead due to swapping instructions and paging data.

This is accomplished by maximizing the work done on each processor while processes run concurrently with those on the other processors, thus maximizing the PUE. Figures 3 and 4 provide examples of hierarchical resources and processes that help simplify such designs.

The instruction language supports hierarchies of rule structures, where looping and complex IF ... THEN ... ELSE statements are flattened - no nesting. What is known as *Waterfall* or *Fall through* code is gone - without GOTOs. These properties dramatically simplify design of complex algorithms. They lead to substantial increases in both understanding and run time speed - on single as well as parallel processors. We note that there is no global data in this system, and all data is automatically referenced by pointer.

When building complex software, human translation is simplified if a language supports obvious representation of physical behavior. Redundancy in a language, e.g., English, supports ease of understanding, and the likelihood that information will be communicated correctly to another person, see [10] and [15]. The examples in Figures 3 and 4 are extracted from large detailed simulations of Packet Radio networks. With hierarchical data structures like those shown, one can represent complex algorithms associated with physical systems with ease. Actual systems may entail more complex resources and processes than those shown, but are easily understood by application experts.

The Separation Principle also underlies visualization of software architectures using engineering drawings. Resources are depicted as ovals in architectural drawings as illustrated in Figure 5. *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, state vector B for dedicated use, and shares state vector C with transformation 2. Therefore, Transformations 1 and 2 are not *independent*.

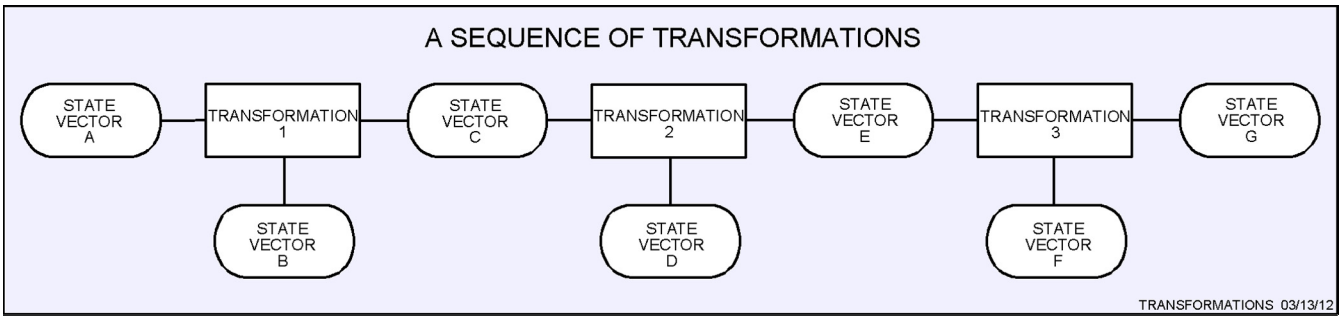


Figure 5. State vectors and transformations.

Properties Of Independence

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. 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; in that case they may be *temporally independent.*

SOFTWARE ARCHITECTURE

As illustrated in Figure 6, 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 6 shows a library module, PROPAGATION_PREDICTION, 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 7. 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 important software architecture requirements. These language properties are critical to simplifying the understandability of large complex software systems.

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 takes maximum advantage of concurrency on a parallel processor. Picking the best set of state spaces is key to solving this problem. Again, best translates to simplicity of transformations and run-time speed.

Having defined *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) are much easier to understand, build, and modify. This is also determined by the *independence properties of the architecture*, i.e. the interconnection of resources and processes. It is the architectural drawings that determine which processes share what resources, eliminating scope rules in the language. This system also eliminates C^{++} type pointers and global data types. All of these unnecessary complexities are in conflict with the design of software for parallel processing.

Instead of attempting to describe architectures at the language level, the problem is separated into the natural hierarchy of drawings and language. To follow this concept further, there are specific module types and resource types, also designated at the drawing level using different colors. These are described below. The simplifications that these visual hierarchies provide become obvious after using them to build large, as well as parallel, simulations/systems.

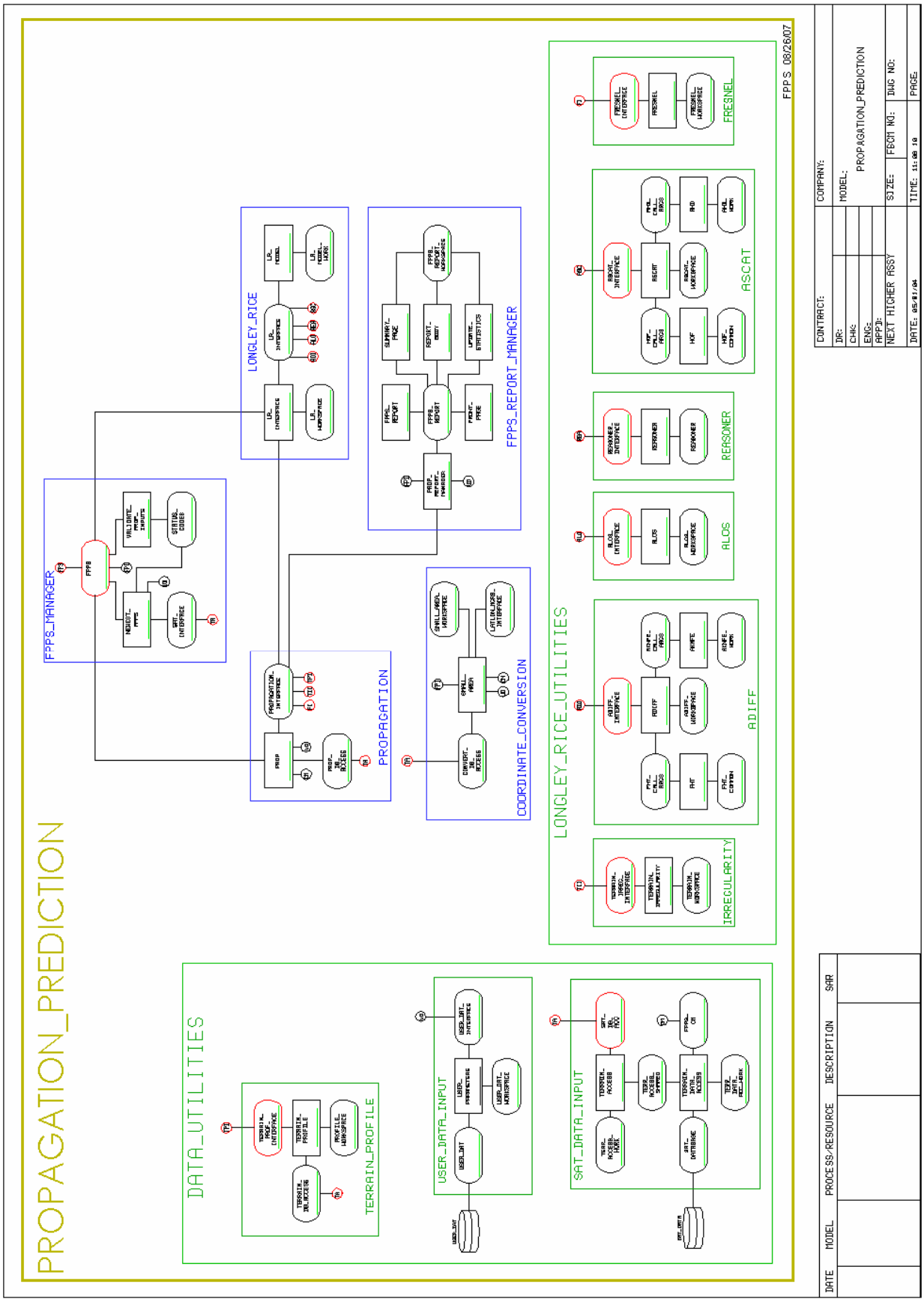


Figure 6. Engineering drawing of a library module.

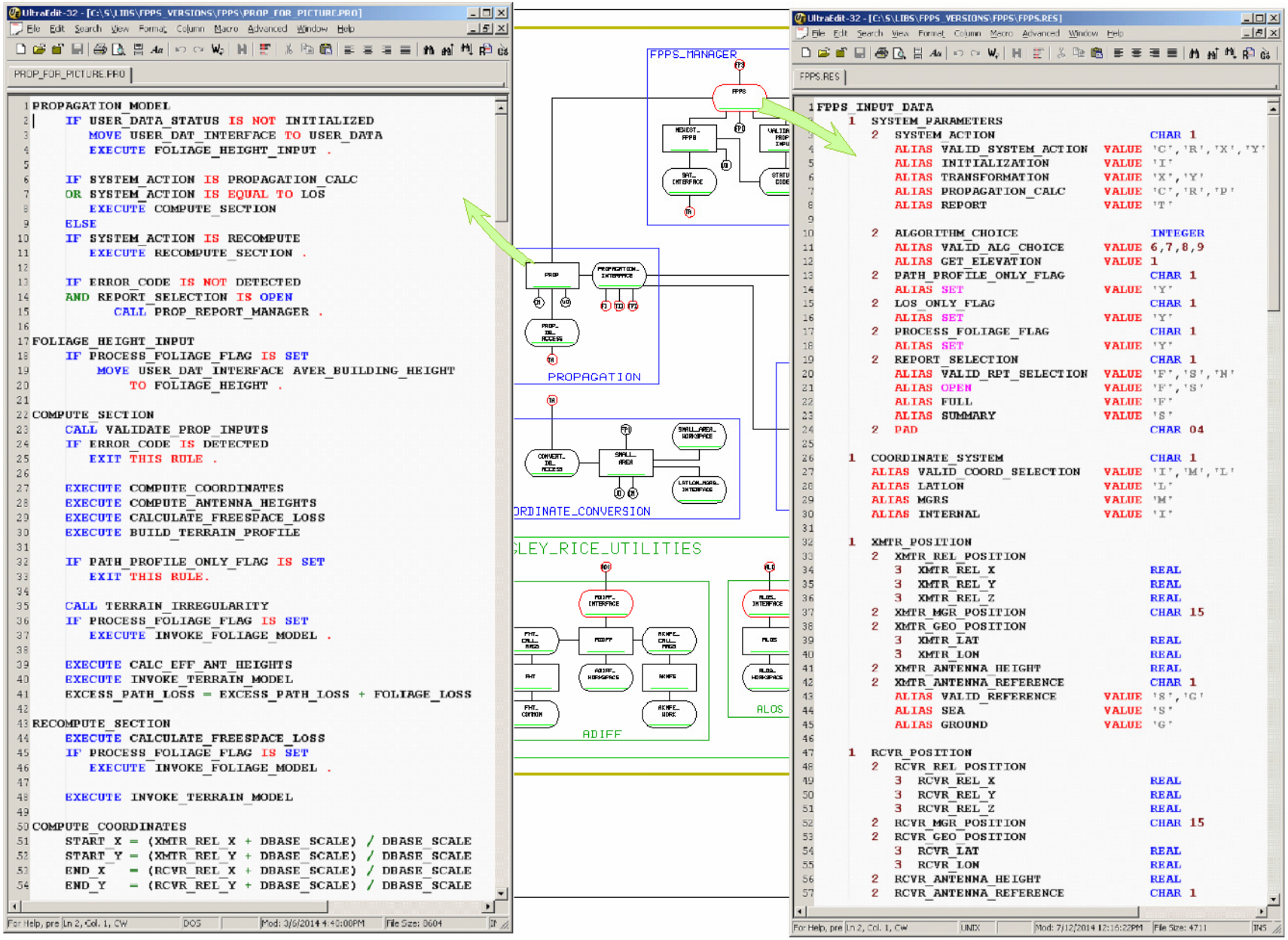


Figure 7. Illustration of editing resources and processes on the drawing.

Types Of Modules

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.

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.
- **IND Modules** - have a brown border. IND Modules only share Inter-Processor (IP) Resources externally - 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. They can only be changed in their own drawing. If they are changed to accommodate a different requirement, that change must be compatible with all processes that use them. Separate copies automatically reside on each processor that uses them.
- **Library Modules** - have a gold border. These are highly protected utility modules that can be shared from different directories and computers, being stored as object modules in special object library files. The source only appears in the directory where they are maintained. Library module Processes 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. Separate copies reside on each processor that uses them.

  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 a new module name. The existing CAD system has a large set of libraries that support various applications, including 3D graphics. These are shared easily.

Instanced Modules

Modules and IND Modules can be instanced. This allows assignment of groups of instances to different processors. This facility is supported by the Control Specification language, providing the ability to assign sequences of instances to different processors to maximize PUE.

Unless one has participated directly in development of VisiSoft architectures, the power of these facilities may take time to comprehend. Once used, it is apparent that the architectural simplifications provided are just as critical to single processor software design. It also becomes obvious that 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. This requires a willingness to compare different approaches based on real measures of speed and productivity.

Timing & Synchronization Of Observations, Actions, And Outcomes.

Software systems running on parallel processors typically represent Objects / Entities that:

- Operate Concurrently
- Act / Interact Based On factors such as:
  - Gravitational Forces
  - Electro-Magnetic Forces
  - Observation of states of other objects
    e.g., position, velocity, transmissions, other actions

Models of these systems must meet, support, and represent the required operational properties of the application with sufficient accuracy. This generally implies:

- Implementing decision processes that produce the desired behaviors of the application.
- These decision processes are typically implemented using complex algorithms operating on complex data spaces.
- These decision processes typically produce responses / actions that are observed by and affect the behavior of other entities in the system.
- The timing and synchronization of - the observations of and responses to - these actions are typically critical to affecting the correct outcomes and responses of the system.
- The representation and implementation of these concurrent observations and actions is critical to achieving the level of accuracy of outcomes required by the application.

This implies maintaining consistency and coherency of data exchanges while operating concurrently on parallel processors.

Capabilities Needed To Support The Above Requirements

Models of entities must be able to:

- Obtain information describing the behavior of other models that are running concurrently.
- Provide information describing their own behavior that can be obtained by other models running concurrently.
- Observe the reactions of other models.
- Complete observations and reactions within specified time frames.

Physical systems can perform these functions while operating concurrently. With the right facilities, these systems can be represented by models operating concurrently on parallel processors.

Facilities To Support The Above Requirements

Software facilities must exist that provide the ability to:

- Schedule events to occur in the future that represent time to complete actions.
- Release information to other models while running concurrently.
- Directly access information from other models that are running concurrently.
- If necessary, wait for information needed to make decisions.

Resource Types

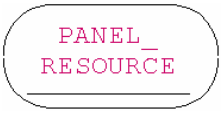
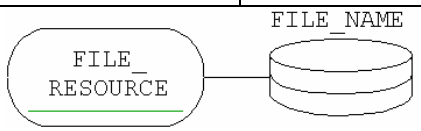
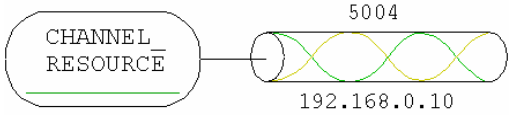
Selection of the type of resource to be shared is a critical architectural decision when designing complex software systems. Resource types support simplicity of the architecture - above the code, invoking substantial facilities that are built into the VisiSoft CAD environment. Designers can create or modify resource types using buttons and panels in the Visual Development Environment (VDE). Panels provide the ability to explicitly specify the types of resources and modules desired, and enter the corresponding information required for a given type. Each resource must be defined explicitly as one of the sharing types in Table 1. This is another illustration of the critical facilities required to simplify complexity when designing architectures of parallel processor software systems.

When building complex software systems, it is important to identify resource types at the engineering drawing level. This is illustrated in Figure 6. Equally important are specific types of modules, also designated at the drawing level using different colors as described below. Module types are used to create hierarchies and to differentiate how they are used and accessed. Depending upon the module type and what is required architecturally, modules must share specified resource types.

To assess parallelism, one must understand the property of module independence. Two modules are *spatially independent* (independent of time) if they share no data (resources). From an application standpoint, these modules have no direct influence on each other.

Two modules are *temporally independent* if they only influence each other at specified points in times. A typical example is when using differential equations to represent dynamic systems. Computer solutions to differential equations typically assign a ΔT time step that is small enough to ensure sufficient accuracy of solutions when compared to actual tests (ΔT may vary for nonlinear systems). This implies that equations are solved at successive time steps, $(T + \Delta T)$, so that changes in one part of a system cause sufficiently accurate effects in another part. Information on changes can be exchanged at the beginning or ending of each time step, affecting calculations within the desired time step.

Table 1. VisiSoft Resource Types.

	Resources hold state data that may be organized in a hierarchical manner. They can be shared by several processes or "dedicated to" a single process in a single task. Connection to either a file or communications channel requires the resource to be dedicated to a single process (described below).
	A resource with a memory template, typically used in a utility or library module. Access to the actual data is provided automatically by a pointer to a resource attached by connector. Shared Alias resources are outlined in red.
	Local Inter-Task resources allow a family of tasks to share data. VisiSoft controls OS level memory management. Local Inter-Task resources are used when a task is responsible for "STARTing" another one that shares the same local Inter-Task resource. Local Inter-Task resources are outlined in green.
	Global Inter-Task resources are similar to Local Inter-Task resources, being 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.
	Inter-Processor (IP) resources are used to share data between Independent (IND) Modules running on different processors in the same task on a parallel processor. IP resources are outlined in blue
	IP Access resources are used to ACCESS data from IP Resources in different IND modules on different processors in the same task running on a parallel processor. IP Access resources are outlined in gold.
	PANEL resources support graphical panel interfaces for input and output of information, which can include icons, check-boxes, scrolling lists, etc. The contents of a PANEL resource are created and modified using the Panel Library Manager (PLM). PANEL resources may be viewed, but not changed with a text editor. PANEL resources are labeled using Red text
	HLA resources support 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. This resource type is used as an HLA Interface.
	A resource describing one or more records on the file to which it is attached. The FILE_NAME identifies the name of the file to be accessed.
	A resource describing the packets and 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.

† Used in military simulations running together in physically separate network connected sites.

More generally, modules are *temporally independent* if they share information on a synchronized basis. This requires a facility where only one module can copy the information to be shared into a resource held for other modules. Other modules can copy that information into their own resources making the information available for follow-on calculations. System level states may be defined to ensure synchronization. These states are shared by pairs of IND Modules. The module changing the resource sets a state to indicate it has updated the resource shared by the pair. The module reading that resource resets it showing that the latest copy has been read. Just as the independence of the real physical systems they may represent, neither module need stop to wait for the other, thus maximizing PUE. An example of such a physical system is people exchanging email.

Use Of Facilities That Support The Requirements

Designing systems that use the above facilities to run modules concurrently on parallel processors requires an in-depth knowledge of the particular application being developed. Only application experts will know how to use the above facilities to deal with the timing and synchronization necessary to ensure proper operation of such systems. Therefore, it is imperative that these facilities be easy to use by those experts.

TIME SYNCHRONIZATION AND SPEED

Given the temporal independence condition, changes in one IND module that affects another on a different processor will be reflected with sufficient accuracy as long as the effects are resolved within a user specified ΔT time period. This requires the ability to *synchronize information exchanges*. This implies that the times when: (1) a sending process updates its IP resource copy; and, (2) the receiving process reads it - both fall within an allowed ΔT. Synchronization is accomplished when the shared IP Resource is exchanged within successive ΔT time periods.

In typical applications, speed is most important, determining whether application run-time constraints are met. When applications impose such a constraint, the optimal design problem is to minimize the number of processors required to meet that constraint. When time constraints can be met using a smaller number of processors, speed will rise exponentially just due to the smaller footprint, dramatically reducing power requirements as well as floor space.

Tests have shown that applications built in VisiSoft can increase speed by 2 orders of magnitude - compared to typical software approaches - *just on a single processor*. When using a parallel processor, reducing the number of processors by only a single order of magnitude can dramatically increase the speed further. Adding the PUE and other multipliers, *it is not unusual to achieve 3 to 4 orders of magnitude increase in speed using VisiSoft on a parallel processor*.

PARALLEL PROCESSOR SOFTWARE DESIGN POINTS

To solve parallel processor software problems, one must take maximum advantage of the inherent parallelism in a particular application to minimize running time. Figure 8 provides an expanded depiction of the steps required to map parallel application requirements into a software space. This requires a software environment that simplifies the design effort for application

experts. As described above, this environment must also support ease of translation of the software design into a hardware environment that maximizes understanding as well as speed. *Much of the burden of design and implementation is then shifted from the developer to highly sophisticated architecture and language translators* - using the computer. As described by Peter van der Linden, [17 - pg 64], this is just where the burden should be.

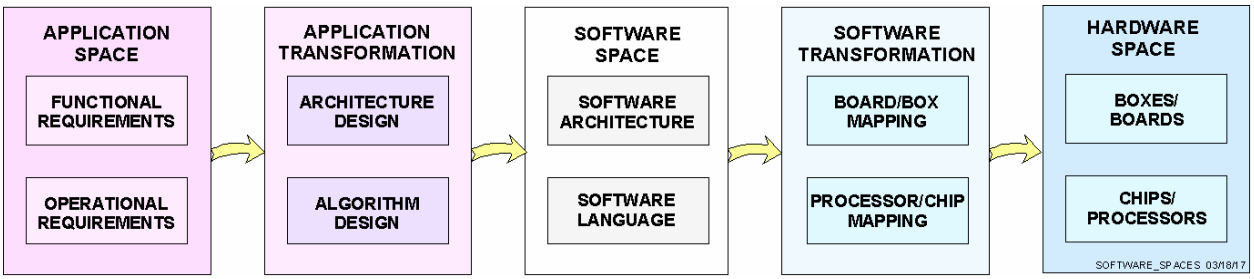


Figure 8. Spaces for translation of application requirements into software and hardware.

The illustration of transformations in Figure 8 implies the following facilities:

- Mapping the functional and operational requirements of an application into a software architecture and supporting algorithms requires: (1) Application experts with knowledge of the complex system and corresponding event spaces; and (2) Graphical facilities for visualization of the software architecture.
- Mapping application space requirements into a software design for a parallel processor hardware space requires special architectural and language facilities. Graphical visualization of the architecture is critical to this process.
- Complex software spaces require the use of deep hierarchical data structures, organized in accordance with the application space - not by data type. These spaces must be easily organized and understood by application experts.
- Complex software algorithms require the use of deep hierarchical rule structures. These must be organized in accordance with the application space - using one-in one-out control structures. The language must be easily understood by application experts.
- The need for application expertise is required to design Independent (IND) Modules. Special graphical visualization tools are critical to simplify understanding of both the module design and mapping process.
- Mapping Independent IND Modules over heterogeneous spaces can provide substantial improvements in speed. Tests must be run on each IND Module to determine their relative speeds - for processor placement - to maximize PUE.
- IND Module run-time speed measurements are provided automatically using the VisiSoft Parallel OS (VPOS) and may be viewed with bar graphs of IND Module run-times on each processor. Speed is improved by reorganizing module assignments.
- Assignment of IND Module Types across Boxes of chips based on physical connectivity can substantially minimize delays over DMA or networked Channels.

- During initialization, the run-time system provides VPOS with architectural information derived from the development environment. This coupled with the expert user allocation of IND Modules eliminates the need for run-time memory management.
- Cross-processor event states save huge amounts of time by synchronizing the exchange of information between processes that continue to run concurrently.

The above bullets represent some of the many facilities that have been built, tested and proven to work on many projects since 1982. It is hard to see how one can obtain the many orders of magnitude of increased speed without the approach highlighted here.

SUMMARY

Design of complex application systems that require parallel processing to meet speed requirements is difficult using current approaches. This paper briefly skims aspects of system design to illustrate some of the top level concepts that must be applied to solve this problem. It is intended to demonstrate an architectural perspective that equates to similar engineering fields, e.g., aeronautical, architectural, electrical, etc. The skills required to design skyscrapers require higher level facilities for designing large complex structures. These skills and facilities are not needed to design simple buildings. Learning to program small problems using different languages is akin to learning the architectural design of small buildings. Large complex systems require a hierarchy of architectural facilities that reside above the code.

This paper only covers top level concepts and facilities that simplify achieving high performance computing. They are part of a system that has evolved building hundreds of large simulations since 1982. This system ensures synchronization of modules running concurrently on parallel processors so they do not have to stop to exchange information. Explicit passing of control to processes on different processors is slow and unnecessary. Scheduling processes to run in future ΔT time periods, or scheduling a process given that an event has occurred on another processor eliminates time wasted waiting for events or testing changes in states.

The facilities built into the VisiSoft Parallel OS (VPOS) support direct synchronization, and eliminate stack management, memory management, and especially coherency management, saving huge amounts of time. VPOS contains accurate clocking, providing measures of processor utilization using graphical representations and supporting optimization of independent module-processor assignments. The results of careful testing and design have produced performance improvements of three to four orders of magnitude for typical parallel processor applications. Tests of these results can be reproduced independently.

The visual simplifications provided by these architectural facilities become more obvious as they are used. Without realization of the importance of the properties of independence and understandability, this architectural approach would have never been developed.

The purpose of this paper is to show that complex automation problems require software languages that support numerous architectural requirements as well as ease of understanding. As in other engineering disciplines, it is the architectural technology that is required to design large complex systems and simulations. This is particularly true when designing systems to be run on parallel processors. Without the VisiSoft architectural facilities, it is hard to understand parallel processing problems let alone solve them.

We recognize that this represents a major paradigm shift, so that gaining acceptance by early adopters would likely be on new project development. But when one examines the rapid accumulation of savings on existing projects, and the ease with which existing software is translated into a much more understandable system, the conversion of existing projects becomes obvious. In any event, we believe that with fair and independent experimentation, the shift is inevitable.

REFERENCES

1. Anselmo, Donald and Henry Ledgard, Measuring Productivity in The Software Industry, *Communications of the ACM*, Vol. 46. No.11, Nov 2003.
2. Anselmo, Donald, *Why Software Productivity Has Not Improved*, Software Summit, Washington, D.C., May 2004.
3. Armour, Phillip G., Software: Hard Data, *Communications of the ACM*, Vol. 49. No.9, Sept. 2006.
4. Beyer, Kurt W., *Grace Hopper and the Invention of the Information Age*. Cambridge, MA: The MIT Press, (2009). ISBN 978-0-262-01310-9.
5. Cave, W.C., et.al, *A Disruptive Solution to the HPC Problem*, Visual Software International, Spring Lake, NJ, September, 2014. http://www.VisiSoft.com/PDF_Files/DisruptiveSolutionToHPC.pdf
6. Cave, W.C., et.al, Software Theory For Parallel Processors, Visual Software International, Spring Lake, NJ, May, 2017. http://www.VisiSoft.com/PDF_Files/SoftwareTheoryBook.pdf
7. Groth, R., *Is the Software Industry's Productivity Declining?, IEEE Software*, Nov/Dec 2004.
8. Kambayashi, Yasushi and Henry F. Ledgard, "The Separation Principle - A Programming Paradigm" IEEE Software, March/April 2004
9. Krishnadas, L.C., EE Times Article, Jan 17, 2008.
10. Ledgard, H., et al, "The Natural Language of Interactive Systems," CACM No. 10, October 1980, pp 556-563.
11. Ledgard, Henry F., *The Emperor with No Clothes*, Communications of the ACM, Oct 2000.
12. Merritt, R., Multicore Puts Screws to Parallel Programming Modules, *EE Times On Line*, Feb 15, 2008.
13. Parnas, D., "Education for Computer Professionals," IEEE Computer, January 1990, pp 17-22.
14. Poore, Jesse H., A Tale of Three Disciplines and a Revolution, *IEEE Computer Society*, Jan 2004.
15. Shannon, C.E., A Mathematical Theory Of Communication, BSTJ, Vol.27, pp 379 & 623, Jul & Oct 1948.
16. Stroustrup, B., "What is Object-Oriented Programming?" (1991 revised version). Proc. 1st European Software Festival. February, 1991. http://www.public.research.att.com/~bs/whatis.pdf
17. van der Linden, P, Expert C Programming - Deep C Secrets, SunSoft Press - Prentice Hall, Englewood Cliffs, NJ, 1994.