OVERVIEW

Various papers describe the need for an engineering approach to overcome barriers when dealing with increased software complexity, see [1], [4], [12], [13], [17], [31] and [33]. This is particularly true when dealing with applications requiring high speed multipliers using parallel processors, see [21], [25], and [36]. Two papers, [7] and [28], call for Computer-Aided Design (CAD) tools for software, and a corresponding engineering approach to software similar to that used in hardware design.

Software complexity depends directly on the application. The hypothetical questions in the bullets below offer examples.

- Is it a web-based social networking application - akin to art & poetry, where the end user requirements depend upon personal feelings and taste?
- Or must it support carefully defined High-Tech requirements, e.g., when engineering real-time communication and control systems, where the software is highly complex?
- Is performance important? Are transaction processing times against large complex databases critical? Must parallel processors be used to meet speed requirements?

Elements of the development environment required to support end user requirements are shown in Figure 1. One must know what can the user afford, what are the user’s time and cost constraints, and what is the competition? Reviewing the applications described in the above bullets, it should be apparent that the last two are totally different from the first in terms of their required level of software complexity. We are concerned here with complex applications, particularly those requiring the use of parallel processors to meet run-time speed constraints.

As applications become more complex, they place a heavier burden on the Development/Support Environment, especially using parallel processors. This burden directly affects developer productivity, and control over the growing complexity of a large software system. Understandability is a critical aspect of complex systems where developers require direct assistance from subject area application experts. Linearization of complexity (versus increasing nonlinearly with size) requires a system to be decomposed into modules that are maximally independent. To achieve high levels of understandability and independence requires that both the code and the architecture be organized into layered hierarchies. The higher the complexity, the deeper the layers of hierarchy.

Implementing understandability and independence depends directly on the language used to produce the code. It must be highly readable by subject area experts and support substantial layers of hierarchy of both data and instructions. The language must also support architectural facilities that provide direct recognition of the connectivity of modules through visualization. These simplifications can only be achieved when the language is tailored to the architectural properties necessary to create an easily understood hierarchical organization.

† The authors are with Visual Software International - www.VisiSoft.com
These goals require that engineering principles be used to compare different approaches to improving development productivity and run-time speed. This can only be accomplished with considerable experiment and measurement, see [3]. Results of experiments are provided below. They support basic principles affecting software productivity. These principles are the basis for VisiSoft, a CAD system that applies directly to the design of complex software for parallel processors. The productivity improvements resulting from this CAD system are obvious.

**Understandability**

As systems become more complex, they become harder to understand, making it imperative that systems be designed to be easily understood. *Understandability* of system design is a property that can be measured by tracking productivity in development, test and production. Learning how to take fair measures requires years of experience working with different types of systems. Understandability can also be measured in terms of the time and effort required by people new to a project to contribute enhancements.

As shown by Ledgard, [23], languages used to specify complex algorithms play a major role in the ability to share understanding. This is inherently a communications problem as described by Shannon, [30]. The language used to transfer information plays a major role in ensuring its correct reception. The English language is known for its redundancy, a major factor in transferring understanding. This is evident in the design of COBOL, the most productive language for information processing. As declared by Grace Hopper, world-wide expert in programming language design (including COBOL and CMS-2), programming languages must be easy to understand - and read like English, the accepted international language, see [6].
As system enhancement increases complexity, one must maintain ease of understanding of the details involved. Understandability must be constantly improved so that people with less experience, or those concerned with only part of the system can be productive. This implies that a system can be decomposed into parts that are relatively independent.

**Independence**

Hardware engineers are concerned with reliability and maintainability of modules composed of parts that can wear out or be damaged. This requires the ability to replace a module with minimal effects on the rest of the system. To do this, modules must be designed to be *independent*, allowing a module to be redesigned with minimal - if any - changes to other modules. Independent modules can be designed and built independently, linearizing the increase in complexity as systems grow in size. This has a major effect on productivity in the original design phase, and is even more important when enhancing the system design.

Independence implies that modules are isolated, i.e., they are not connected. For modules to be maximally independent, they must be minimally connected. In the case of software, two modules are *spatially independent* if they share no data, i.e., they are not connected. In addition, there may be periods of time when only a single module has write access to shared data and no others have any access to that data. There may also be periods of time when modules only have read access to the shared data. In both of these cases, the modules are *temporally independent* during those time periods.

This principle requires a major change in the approach to developing software. Its value is most apparent when building software for parallel processors: Modules must be independent to run concurrently on separate processors. When modules are independent, they are also much easier to understand. Figure 2 illustrates that understandability is *not* orthogonal to independence, since understandability increases with independence. These two interrelated properties are key to dealing with complexity.

![Productivity Vectors](image)

Figure 2. The critical factors affecting the ability to deal with complexity.

**Data Spaces**

When solving complex mathematical problems, one learns to formulate problems based on spaces that simplify the solution approach. Selection of a good space typically reduces the complexity of the equations and the corresponding time it takes to solve the problem. A simple example is the equations of motion of a particle moving in a spherical orbit using a Cartesian coordinate system versus that using spherical coordinates - a great simplification.
Software spaces are determined by the databases used to support the instructions (transformations) that implement the dynamics of an application. One can think of the entire database as the overall software space. This space is broken into subspaces that support specific actions or transformations implemented by sets of instructions.

**Software - An Extension Of Mathematics**

As shown by experiments, [5], [8], and [19], simplifying software design is most easily accomplished with the basic principle used to solve complex math problems. This is to find the space that provides the greatest simplification of the equations. This approach is common when formulating solutions to large sets of differential and partial differential equations. The most critical property underlying this approach is the use of the best set of independent coordinates.

For large problems, selecting the best vector space provides a great simplification of the equations with substantial gains in speed. This approach becomes more obvious as the problem becomes more complex. For many problems, multiple spaces can be used to represent solutions. The best space provides maximum simplification, typically achieving maximum solution speed.

What are the equivalent transformations required for software? If the application is mathematical, e.g., using discrete-time or continuous-time models, the solution is likely mathematical and similar. Software applications typically require actions based upon events as they unfold. Discrete Event Simulation provides significant insights into this problem. Although time is still a basic coordinate, actions jump to the next scheduled event. The difference is that actions typically depend upon complex decision processes, e.g.,

```plaintext
IF A IS TRUE ... EXECUTE RULE_A ... ELSE IF B IS TRUE ... EXECUTE RULE_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 may be used in a time based model or a real-time system. The need to schedule a future event is determined when a particular action is taken. Scheduling the event at the time of the action eliminates the need - at the future time - to loop through a sequence checking values, or to check a list of possible actions to determine if some flag has been set. Each of these later approaches can waste considerable time. Instead, the event is performed when popped at its scheduled time. One must also be able to cancel events prior to their scheduled time.

**Simplification of Parallel Processing Software**

A software development environment for parallel processors must support designers faced with creating architectures of independent modules based upon knowledge of the inherent parallelism in a system. This implies that two or more processes† (groups of instructions) can run concurrently on separate processors. This requires that the processes must be independent, i.e., they share no data. This requirement led to the approach described here, where the decision was made to separate data from instructions so that the independence property could be established and tracked easily in the development environment. To accomplish this, one must separate data from instructions at the design interface level so that module independence (or lack of it) is clearly recognized.

† Process as used here is similar to an assembler language subroutine that contains only instructions that reference only external data (by pointer). It has no relation to a UNIX process.
The Separation Principle

The underlying principle supporting modularity and visualization of software is the separation of data from instructions at the language level. Defined in 1982 in the design of the General Simulation System (GSS), [18], this has become known as the *Separation Principle*, [20]. In GSS, separate languages are used to describe data structures (*Resources*) and rule structures (*Processes*).

As described below, this led to Generalized State Space, an extended mathematical framework for implementing GSS. This approach has bred solutions that provide order-of-magnitude reductions in run time on a single processor while substantially simplifying the use of parallel processors with high Processor Utilization Efficiencies (PUEs), see Chapters 17 and 18 in [10].

Generalized State Space

Generalized State Space capitalizes upon the concepts of the State Space framework used in control theory by extending the mathematical definitions of vectors and transformations, see [16] and [29]. Based on Shannon’s theory of binary systems, we introduce the concept of a *Generalized State Vector*. Instead of restricting a vector to numbers, it can take on states described by words. For example, the state LIGHT may take on the values RED, YELLOW, or GREEN. In addition, transformations on a state need not be restricted to typical mathematical operators. For example, we may want to say:

```
IF LIGHT IS YELLOW, SET LIGHT TO RED
```

a *Generalized Transformation*. Given this facility, one may view computer software as consisting of generalized state vectors (data) and transformations (instructions).

Using the Generalized State Space framework, the Separation Principle is achieved by storing all data in *Resources* that generally contain hierarchical data structures. Resources are depicted as ovals in architectural drawings as illustrated in Figure 3. *Processes*, defined above, contain instructions in the form of hierarchical rule sets. They are depicted as rectangles.

![A SEQUENCE OF TRANSFORMATIONS](image)

Figure 3. State vectors and transformations.

In Figure 3, each transformation has a dedicated state vector and shared state vectors. 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. 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.

**Generalized State Space - An Extension Of Mathematics**

The approach described here defines Generalized State Space as the framework for creating software architectures. Development of this framework was motivated by multiple factors. One was the requirement to embed decision algorithms within mathematical models. For example, communication system designers want to embed natural language conditions within models as shown below.

```
IF MESSAGE_TYPE IS CONTROL
    THEN EXECUTE CONTROL_RULE
ELSE
    IF MESSAGE_TYPE IS DATA
        THEN EXECUTE DATA_RULE
```

Whereas State Space provides a framework for representing most any type of dynamic system using numbers and mathematical operations, Generalized State Space extends this facility, incorporating data spaces and software decision algorithms, providing an extended mathematical framework.

**Modularity**

Another property of engineered systems that reduces complexity is modularity. An example of a system decomposed into modules is illustrated in Figure 4. This example portrays the typical spaghetti-like code when showing the interconnection of resources and processes. Without the drawings, sharing of data is often driven by the desire to save memory, and huge sets of dependent interconnections are common. The typical complaint in such systems is that a modification of one part of the software affects other parts that were thought to be independent, see [14]. Systems are best designed and maintained when they are decomposed into independent modules as illustrated in Figure 5. This typically uses more memory - currently the most abundant resource. Most importantly, it allows modules to be refined or replaced with minimal, if any, changes to the rest of the system.
Figure 4. Telephone network model - a pre-drawing version.
Figure 5. INTER_OFFICE_NETWORK Model using RTG.
Modularity implies the ability to decompose a system into parts that are reasonably independent. This generally requires the knowledge of a subject area expert on the system to be developed. Examples are complex applications to be used by people who are unfamiliar with software development, but are well versed in the requirements for the application system being developed or enhanced. It also requires grouping small data declarations into large hierarchical data spaces within in a single resource.

SOFTWARE ARCHITECTURE

Software Architecture - Modularity & Independence

In most engineering fields, 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.

Figure 5 illustrates a decomposition that follows the organizational lines of the application. This may not always be the case. For example, an interactive system involving graphical interfaces will use graphics modules as the direct interface to the user. These graphics modules will be interfaced to modules that handle input or output to and from the application system. Higher level input/output modules will be part of the overall application architecture that has been decomposed based upon the application. Graphical interface modules may be interfaced to multiple application modules. Both types of modules may be designed to be maximally independent. And both the architectural decomposition and the design of the individual modules will most likely require subject area expertise.

Hierarchies

Figure 5 illustrates the use of a hierarchical module to design the system. Going back thousands of years, organizations are best controlled using hierarchies (Without hierarchical organizations, the military would be in chaos). Hierarchical structures are a critical property of software, both in architecture and language. They support the specification of complex data spaces that are used to simplify complex algorithms. The number of levels in a hierarchy must be sufficient to push down the complexity, making the organization of the system easy to understand.

As illustrated in Figure 5, 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 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 are edited directly on the drawing as illustrated in Figure 6. The languages do not permit the declaration of scope rules. It is the architecture that determines how data is shared, and the corresponding module independence. Most important, the languages are designed to provide for deep hierarchies in both data structures and rules to support the critical software understandability requirements defined above. Without these language properties, understandability of complex software is difficult.
Hierarchical Modules

Given that a system is to be decomposed into independent modules, one must create an architecture of interconnected modules. As the system becomes complex, it is necessary to create hierarchies of modules, where the number of levels in the hierarchy will be determined by the size and complexity of the system. The manner in which the decomposition of modules is done will depend directly on the application. Design of the overall hierarchy as well as the sub-hierarchies will determine the understandability of the overall system and the time to develop and enhance a reliable product. This design will depend heavily upon the nature and functionality of the application system itself. This depends upon critical contributions from subject area experts.

Visualization

Most engineering fields (e.g., Architectural, Aeronautical, Electrical, Mechanical, etc.) would be at a huge disadvantage without engineering drawings, the precise descriptions of connectivity. Software is no different. But one has to have sufficient experience using them to understand their importance. Engineering designs also require written (language) specifications. Typically, the architecture versus language crossover point is obvious, as it is in software. But without languages that support deep hierarchies of data structures and rules, both independence and understandability are lost. The languages must also support the modularity and visualization of software architectures, else the advantages of engineering drawings cannot be realized.

LANGUAGE AND INFORMATION THEORY

The more information one has to make a decision, the more likely a good outcome. If information is misunderstood, the probability of a poor outcome increases. The two major objectives of communication systems are reliability and speed. Fast and reliable transfer of information is the goal of information theory, as evolved by Shannon, [30]. The basic principle is that: Reliability of information transfers is increased by adding redundancy (i.e., additional data). This is used to write and read computer memory where code bits are added to the data being stored, decreasing the probability of error when reading it. In wireless communications, one may double the size of the original data stream to ensure reliable transfer. One may send the same message twice, or use additional words, such as articles, adjectives, or adverbs. English is considered to have a high degree of redundancy compared to other languages, implying it is more likely that information is transferred reliably - and key to the survival of its users.

As stated by Bjarne Stroustrup, creator of C++: “English is arguably the largest and most complex language in the world (measured in number of words and idioms), but also one of the most successful,” see [34]. It dominates the world of free trade. Considering the small islands where it originated, its survival is attributed to the success of its users - thanks to its reliability.

Studies comparing interactive languages have shown that errors increase as statements move from good English to a more terse form, see [23]. Comparisons of COBOL, FORTRAN and C-based languages will typically derive the following conclusions: COBOL is verbose; FORTRAN is fair; C-based languages are terse (an objective of the principle designer).
Generalization To Software

Figure 7 illustrates the problem of developing software to solve complex problems. Designers must be able to translate application requirements into a solution using a user-friendly language (software space). This language must be designed for human understanding, making it easy for subject-area experts to map the application into a software space that simplifies the design. Figure 8 illustrates the language used to define a hierarchical data space.

![Diagram](image_url)

**Figure 7.** The environment of software language translators.

<table>
<thead>
<tr>
<th>RESOURCE NAME: TRANSEIVER</th>
<th>INSTANCES: TRANSMITTER RECEIVER</th>
</tr>
</thead>
<tbody>
<tr>
<td>GENERAL_PARAMETERS</td>
<td></td>
</tr>
<tr>
<td>1 TRANSMITTER_POWER</td>
<td>REAL INITIAL_VALUE 100</td>
</tr>
<tr>
<td>1 RECEIVER_THRESHOLD</td>
<td>REAL INITIAL_VALUE 120</td>
</tr>
<tr>
<td>RADIO</td>
<td></td>
</tr>
<tr>
<td>1 TRANSCEIVER</td>
<td>STATUS TRANSMITTING RECEIVING IDLE OFF</td>
</tr>
<tr>
<td>1 LOCATION</td>
<td></td>
</tr>
<tr>
<td>2 X_POSITION</td>
<td>REAL</td>
</tr>
<tr>
<td>2 Y_POSITION</td>
<td>REAL</td>
</tr>
<tr>
<td>2 ELEVATION</td>
<td>REAL</td>
</tr>
<tr>
<td>1 ANTENNA_HEIGHT</td>
<td>REAL</td>
</tr>
<tr>
<td>1 ANTENNA_GAIN</td>
<td>REAL</td>
</tr>
<tr>
<td>RECEIVER_CONNECTIVITY_VECTO</td>
<td></td>
</tr>
<tr>
<td>1 POWER_AT_RECEIVER</td>
<td>REAL</td>
</tr>
<tr>
<td>1 TOTAL_NOISE_POWER</td>
<td>REAL</td>
</tr>
<tr>
<td>1 CONNECTIVITY_MATRIX</td>
<td></td>
</tr>
<tr>
<td>2 PROPAGATION_LOSSES</td>
<td></td>
</tr>
<tr>
<td>3 TERRAIN_LOSS</td>
<td>REAL</td>
</tr>
<tr>
<td>3 FOLIAGE_LOSS</td>
<td>REAL</td>
</tr>
<tr>
<td>3 TOTAL_LOSS</td>
<td>REAL</td>
</tr>
<tr>
<td>2 SIGNAL_POWER</td>
<td>REAL</td>
</tr>
<tr>
<td>2 SIGNAL_TO_NOISE_RATIO</td>
<td>REAL</td>
</tr>
<tr>
<td>2 LINK_DELAY</td>
<td></td>
</tr>
<tr>
<td>2 LINK</td>
<td>STATUS GOOD FAIR POOR</td>
</tr>
<tr>
<td>TRANSCEIVER_RULES</td>
<td>RULES GOOD RECEIPTION CONFLICTING_RECEIPTION CONFLICTING_BROADCAST</td>
</tr>
</tbody>
</table>

**Figure 8.** Building hierarchical data structures using the Resource Language.
Hierarchical Data Spaces

Translation of human-oriented languages into binary code is done by the computer. In addition, complex data structures are more easily understood when put into a hierarchy, see Figure 8. With languages designed to simplify mapping application solutions into software space, the computer translation becomes much more complex. But that is the purpose of this CAD system, and exactly where the burden should be, see [35], pg 64.

Hierarchical Transformations

Figure 9 shows a VisiSoft Process. The hierarchical organization applies directly to the simplification of complex instruction sets using one-in one-out control structures, see [26].

Figure 9. Building hierarchical rule structures using the Process Language.

In VisiSoft, which processes share what data is determined solely by the architecture, not the code. This includes the manner in which data is shared. There is no global data in VisiSoft. Also, all data is automatically accessed by pointer.
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 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, see [8]. 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.

The critical property determining complete and consistent results of the transformations, and the ability to run concurrently, is that of independence. This implies that transformations are spatially independent when they share no state information. This property - two processes not connected to the same resource - represents the spatial independence of the processes. As shown below, when processes are connected to the same resource but inhibited from running concurrently, e.g., on a single processor or through synchronization, they have temporal independence.

Software Languages To Support Parallelism

Requirements for the resource and process languages were driven in part by factors similar to those motivating the use of tiling in parallel versions of FORTRAN. These factors minimize memory management overhead due to swapping and paging. 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, [5], such organizations are best supported by deep hierarchies of both data and instructions.

Inter-Module Communications

Independent (IND) Modules residing on separate processors must communicate with each other. Design of the resources used to communicate between IND modules must ensure their independence and the ease of understanding of module design.
Communication between IND modules residing on different processors is most easily accomplished using a simplex channel. Two-way communication can use a pair of simplex channels. However, experience with architectural design for parallel processors shows that Inter-Processor (IP) resources are naturally written by only one module, and only read by IND modules on other processors. This is because the results obtained by one IND module are typically communicated in one direction for use by others. This is convenient since, if IND modules are exchanging data, they must be synchronized to run concurrently. Sending and receiving data efficiently time-wise is best done using separate one-way channels, allowing synchronization to be performed easily. Using VisiSoft, synchronization is done automatically.

Deep hierarchies allow large complex data structures to be moved in a single instruction fetch, with all of the individual fields directly available to processes that use them, see Figure 10. This provides order of magnitude improvements in single as well as parallel processor speeds.

<table>
<thead>
<tr>
<th>RESOURCE: INDEXED_MESSAGE_TABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MESSAGE_TABLE QUANTITY(3)</td>
</tr>
<tr>
<td>1 MESSAGE_INDEX INTEGER</td>
</tr>
<tr>
<td>1 MESSAGE_ELEMENT QUANTITY(13)</td>
</tr>
<tr>
<td>2 UNIT_ID INTEGER</td>
</tr>
<tr>
<td>2 SLOT_ID INTEGER</td>
</tr>
<tr>
<td>2 MESSAGE_INFORMATION</td>
</tr>
<tr>
<td>3 MESSAGE_TYPE STATUS DATA_OUTPUT</td>
</tr>
<tr>
<td>3 MESSAGE_STATE QUANTITY(7) STATUS EMPTY, FULL</td>
</tr>
<tr>
<td>4 STATE_S</td>
</tr>
<tr>
<td>4 NUMBER_TO_BE_SENT INTEGER</td>
</tr>
<tr>
<td>4 SEQUENCE_NUMBER INTEGER</td>
</tr>
<tr>
<td>4 MESSAGE_ACTION STATUS SEND, HOLD</td>
</tr>
<tr>
<td>4 AGGREGATE_STATE</td>
</tr>
<tr>
<td>5 MESSAGE_STATE QUANTITY(7) STATUS EMPTY, FULL</td>
</tr>
<tr>
<td>4 INDIVIDUAL_STATE REDEFINES AGGREGATE_STATE</td>
</tr>
<tr>
<td>5 SEQUENCED_MESSAGE</td>
</tr>
<tr>
<td>6 GROUP_MESSAGE STATUS EMPTY, FULL</td>
</tr>
<tr>
<td>6 BUDDY_MESSAGE STATUS EMPTY, FULL</td>
</tr>
<tr>
<td>6 QUEUED_MESSAGE STATUS EMPTY, FULL</td>
</tr>
<tr>
<td>6 RESERVED_MESSAGE STATUS EMPTY, FULL</td>
</tr>
<tr>
<td>6 INTERCOM_MESSAGE STATUS EMPTY, FULL</td>
</tr>
<tr>
<td>5 NON SEQUENCED_COMMAND</td>
</tr>
<tr>
<td>6 DATA_INPUT STATUS EMPTY, FULL</td>
</tr>
<tr>
<td>6 USER_COMMAND STATUS EMPTY, FULL</td>
</tr>
</tbody>
</table>

Figure 10. Example of a hierarchically structured Resource.

When building complex software, human translation is simplified if a language supports obvious representation of physical behavior. The examples in Figures 8, 9 and 10 are taken directly from large simulations of complex communication networks. With hierarchical data structures like those shown, one can represent the complex algorithms associated with physical systems with ease. This is illustrated in Figure 9. Actual systems may entail more complex resources and processes than those shown, but are easily understood by subject area experts.

Not shown in Figure 8 are the QUANTITY clauses for TRANSMITTER and RECEIVER. Likewise, the corresponding subscripts are not used in Figure 9. This is because the resource and process pair is part of an instanced module where instances are automatically set at the module level - when a process within an instanced module is CALLed or SCHEDULEd. Moving instance implementation to the architecture level substantially improves the understandability of the code. The real systems do not use instances pointers.
The system described here, VisiSoft, was originally designed in 1982 as a CAD system for discrete event simulation. Underlying this CAD system are three language translators as shown in Figure 11: one for data; one for instructions; and one for run-time control. Control Specifications render the software independent of the OS and platform, eliminating scripts while supporting complex databases, graphics, and allocation of parallel processors. To make it easy for the human to understand, the languages are context oriented, requiring the three computer language translators to be extremely complex pieces of software.

Software Spaces For Parallel Processing

Software modularity was considered in the late 1960s and early 1970s, see for example Gauthier and Pont, [15], and Parnas, [27]. However, the concepts lacked precise definitions, and were soon overtaken by the use of C-based languages (C, C++, C#, Java, etc.) and OOP.

A major drawback of these later approaches is the use of abstractions and data hiding. More importantly, they inhibit the creation and use of large hierarchical data structures. These properties obscure which instructions share what data. The use of inheritance exacerbates this obscurity. Such approaches inhibit the understanding and determination of the property of independence - the key to software architecture for parallel processors. Coupled with the terse and cryptic nature of C-based languages, it is difficult for subject area experts to understand the algorithms, and hard to control the growing complexity of large software systems - independent of using parallel processors. These problems are described by Don Anselmo, Director of Computer Development at Bell Labs when C and UNIX were developed, see [1] and [2].

Prior to the technology described here, simulations of large mobile communication systems were taking 5 to 7 days to run a 2 hour scenario. This led to the major design requirement for VisiSoft, i.e., simulations must run on a parallel machine. As described above, creating software to take maximum advantage of a parallel processor requires that the software be decomposed into independent modules that can run concurrently. This architectural decomposition must reflect the inherent parallelism of the system it represents. The critical element supporting this requirement is the ability to define the data spaces that simplify the design.

Critical Software Architecture Requirements

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 IND modules can run concurrently. This implies creating modules that are independent, i.e., they only share data using IP resources. To determine the independence of processes, designers must be able to easily see which processes share what data (IP resources). This can only be done when the following critical requirements are met:

- Data structures can be organized into the deep hierarchies required to represent the best spaces to implement problem solutions;
- Data is organized into a minimum number of structures shared between processes;
- Designers can easily determine which processes share what data so they can assure their independence properties.
Figure 11. Example of an engineering drawing of a parallel processor simulation.
The above requirements are best met when the data language supports large data structures using deep hierarchies. It also requires an instruction language that supports 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, see [9] and [10].

**Taking Advantage Of Architectural Information At Run-Time**

To take advantage of a parallel processor at run-time, the OS must map threads onto processors to maximize the speed multiplier. A designer faced with generating complex algorithms should not be concerned with this problem. Similarly, a traditional compiler will have little success trying to interpret an architect’s decomposition of modules from the code. Finally, the operating system will not be very successful in determining where to map threads based upon current run-time statistics, especially if they are nonstationary.

Referring to Figure 12, if sufficient inherent parallelism exists in the system, architects can decompose software into large IND modules. As described in [10], threads are contained within IND modules. Because threads in one IND module are independent of those in another, they can run concurrently on separate processors without concern for synchronization.

![Figure 12](Image)

Figure 12. The parallel processor software-hardware environment.

**Temporal Independence And Synchronization**

As stated above, IND modules communicate using IP resources. IP resources may be read or written by any process within the same IND module, but processes outside that module may only read them. Two-way communications is implemented using two IP resources - one in each IND module that writes to that IP resource, with copies to all that must read it.
All IND modules are automatically synchronized in time within a user-specified $\Delta T$ Time Interval by the VisiSoft Parallel OS (VPOS) to ensure that the results are complete and consistent, see [5]. $\Delta T$ is determined by the designer based upon the accuracy requirements on the application results. By synchronizing the release of, and access to IP resources, the temporal independence of IND modules that communicate with each other is automatically ensured by VPOS.

**The VisiSoft Run-Time System**

The architectural information that characterizes inherent parallelism of an application system is contained in databases that support the CAD development environment. A Run-Time System (RTS) is generated from that information to control VPOS calls that assign modules to processors. It also ensures that the resources reside with the processes that use them.

VisiSoft IND modules are typically large and remain on a specified processor, minimizing if not eliminating swapping and paging, and increasing Processor Utilization Efficiency (PUE). However, as processor loads become unbalanced, PUE will fall and one must carefully consider the application level design constraints, typically to minimize the number of processors required to meet a given run time constraint. The VisiSoft CAD system contains built-in measurements and graphical depictions of the PUE, by IND module, so that users can easily assess and balance the loading on each processor.

**FAIR COMPARISONS BASED UPON EXPERIMENT AND MEASUREMENT**

When designing and conducting experiments to determine the best software languages and development environments, one must avoid the potential pitfalls that produce biased results. There are multiple causes. First is the wide range of software applications, from personal to commercial, to industrial, to government and military. The last two categories have invested the largest share of money in parallel processing, but are not nearly as concerned about economics as a small private business.

Selling CAD systems to engineers who want to cut the time to produce designs is significantly different from selling them to programmers concerned about job security. When programmers perceive that the huge investments they have made learning esoteric computer languages are no longer needed, they resist. The result is hardly different from the technology companies that tried to sell robots to Detroit’s auto industry in the 1960s. Robots were effectively banned in the U.S., so their technology developers ended up selling them to Japan.

If properly focused on the economics, particularly the time and cost to build and maintain software, then fair outcomes can certainly be achieved. However, experimenters must beware of the forces that are brought to bear to hold off disruptive technologies, especially those that provide huge cuts in time and cost. As described by Christiansen, [11], and Kuhn, [22], such technologies are often at odds with existing investments, both in the time spent learning to use an inferior approach, as well as financial investments behind the existing approaches.

Conducting experiments to measure the run-time speed of a reasonably large complex software system is much more simple than measuring productivity, but also subject to biases. However, given the proper scientific environment, speed measures are easily accomplished.
REFERENCES


