Understanding Programming Language Semantics For The Sophisticated World

Trong Wu, *Member, IAENG*

***Abstract* — Computer is used virtually everywhere everyday in the world. Before 1990s computer systems are generally used for mathematics, engineering, and business computations. In this period, mainly use *FOTRAN*, *COBOL*, and *PL/1* for computation on mainframe systems. In the last two decades scientists found that the natural world is complicated that has overwhelmed with data that required more sophisticated computation facilities and better languages for computation and that created new computing disciplines. This paper addresses the understanding of programming language semantics that will help user in selection of programming language features for various applications needs and that can help programmers in designing reliable, accurate, efficient, and user-friendly software systems for the sophisticated word.**

***Index Terms*: user-defined types, exception handling, multitasking, communication between program units, and user-defined precision**

I. Introduction

It is a chaotic world, it is a systematic world; it is a confused age, it is illuminate age; it is disordered society, it is organized society. Everyday we are struggling in a complex, intricate, and difficult environment for our life. Computer science can give one abilities to solve a category of complicated, obscure, and oppressive problems. We can take the advantage of computer speed and large volume of storage spaces to add our ability for solving complex problems in the world. This is a very challenge task and interesting assignment to youngsters. It attracts many of them to study computer science.

In October 1962, Purdue University established the first department of computer science in the United States [8]. Since then Computer Science education has become an integral discipline in colleges and universities across the country. Initially, students in physics, engineering, and mathematics were advised to take one course in the *FORTRAN* language while students in the school of business particularly in Management Information Systems ware required to take a course in the *COBOL* language. Students who majored in Computer Science were required to enrolled in both *FORTRAN*, *COBOL*, and *PL/1* courses. It was thought that this would prepare them potentially to work in industrial firms or business companies after completion of their degrees.

The author: Trong Wu is a Professor in the Department of Computer science at Southern Illinois University Edwardsville, Edwardsville, Illinois 62026, U.S.A. His e-mail address: twu@siue.edu; phone: 618-692-4027.

Since then, computer software and its use have spread to nearly every aspect of peoples' lives from work to play. Today almost every one uses computer and software everywhere. Most of the equipment and devices, which use software are designed and manufactured by engineers. Some of this hardware is equipped with *embedded computer systems*, some are operated by a computer system, and some are connected to a computer and give output results to the computer. For these applications, users need not only robust hardware, but also reliable software.

Today, the most serious problems with software products are *expense*, *ease of use*, and *reliability* [12]. Computer scientists not only need to study computer mechanisms and how to increase the productivity and efficiencies of a computer system, but also need to design computer systems, write programs, and execute them. The former is the *state of the practice of software engineering* and the latter is the *state of the art*. Unfortunately, there exists a gap between the former and the latter. The task for computer scientists is to eliminate or narrow this gap. To do this, one may apply the principles of *software engineering* for software design that will make systems more efficient, robust, reliable, and friendlier to use. To implement this, one should thoroughly analyze system requirements, carefully design the system and programs, and then perform various tests including unit tests and system tests. Finally computer scientists need to deliver systems and make sure that they meet their requirements and fit in their environments [14].

However, computer scientists need a good language tool to implement these software engineering principles. Human languages such as English, Chinese, Spanish, Russian, etc. are all ambiguous languages. None of them can be used to communicate with a computer system. Therefore, mathematicians, computer scientists, and linguists must work hard to develop *semantically unambiguous* languages, starting by designing *grammar rules* and then developing the *sentential structures* of the language. A computer program uses such a language to communicate with a computer system to take a set of input data to the computer system, reorganize data objects in the computer system, construct new data objects, and then generate a useful output for the program.

Nevertheless, most engineers have had only one computing course in *FORTRAN* or in the *C* language or both while they were in college and perhaps they believe that *FORTRAN* or the *C* language is enough for their engineering application needs. In fact, the *FORTRAN* language lacks user-defined types that limit its application domain. Moreover, both the *FORTRAN* and *C* languages do not have *predefined accuracy features*, *friendly concurrency facilities*, and *effective exception handling methods*. Today, neither *FORTRAN* language nor the *C language* remains adequate to program in this complex and chaotic world for designing and implementing more sophisticated and reliable systems for biological, physical, and other natural related systems. Therefore, we need understand programming Language semantics thoroughly so that we can deal with natural world computation.

This paper describes some necessary elements in the understanding programming language semantics that is important for the computation of the natural world. It requires *computation more accurate*, *more reliable*, *more precise*, *more efficient*, and *friendlier*. Therefore, we should consider the *extends of application domain*, the application of *accuracy* and *efficient* facilities, the use of *modularization* and *communication*, the *utilization of parallelism* or *concurrency*, and *applying exception handling* and for *reliability critical projects* in this sophisticated word.

II. EXTENDSION OF THE APPLICATIONS TO A SOPHISTICATED WORD

A department store usually consists of tens' of thousands merchandises; it is very difficult to manage them *properly*, *efficiently*, and *cost-effectively*. One of the most effective ways to manage such department stores is to partitioning all the merchandises into departments such as *man's ware*, *lady's ware*, *house ware*, *juniors*, *sports*, *electronics*, *hardware*, *shores*, *bath and bed*, *pharmacy*, etc. Any merchandise in a department store must belong to one and only one department, and department-to-department are discriminates to each other. No merchandises can belong to two distinct departments at the same time. If a piece of merchandise belongs to two departments that must have two stock numbers, that the item will have two entries on the computer printed list, and that will increase the operating overhead for department store. Each department consists of a set of similar or the same kind merchandises and a set of operation rules and policies for manage these merchandises. For example, *hardware* and *electronic* departments have different return or refund policies; some *electronic items* are not allowed customers to test or to tryout. *Pharmacy department* has its operational procedures or rules; medications are classified into *prescription* and *non-prescription*. For any prescription medication is strictly required a medical doctor's signed prescription [26].

In a special season of the year, like Christmas time, most department stores want to make some additional business for this splendor season; they create a new department called "*Santa*;" it is for children to take pictures with the *Santa*. Likewise, in the springtime they create a new department called "*Plants*" by selling saplings, flowers, seedling, soils, seeds, fertilizer, rocks, and other goods for the garden [26].

Today real world projects can be very large and complicated for applications in newly developed domains. These projects can include *hardware projects*, *software projects*, and *firmware projects*. Many engineering projects need to take years or tens' of years to complete them such as the *Yangtze River Three Gorges Dam project* that was launched in 1993 and the water level in the reservoir will reach to 175 meters in 2009, when the project is finally completed [7]; the *Airbus A380 project* that was started in 2000 for worldwide market campaign and made its maiden flight in 2005 [2]; and the *Boeing 777 program* was launched in October 1990 and the first 777-300 was delivered to Cathay Pacific Airways in June 1998 [5]. These projects involved thousands designers, hundreds subcontractors, and tens of thousands manufactures. To implement and manage these projects we need to use many *large-scale computer systems* to process a massive amount of data and programs.

A computer system commonly manages a large amount of data objects. In structure it is similar to a *department store*, with its thousands of items of merchandise, or *a large engineering project*, with hundreds of subcontractors. Therefore, we need to group *data objects* into *types*. *Types* are discriminates to each other, unless a *type* conversion function is applied that can converts from one *type* to another. *Data objects* in a computer system are analogous to *merchandise* in a department store or *devices*, *parts*, and *equipment* in an engineering project. The *association rules* used to manage *merchandise*, *parts*, *devices*, and *equipments* are similar to *respect operations* of *types* in a computer system. To create a new department in a department store is to expand its business; much like creating a *new type* in a computer system enlarges its application domain [26] to *multidisciplinary areas* and to the control of *more complex physical systems*. These new types are called *user-defined types*.

Similarly, management of an engineering project requires subdividing the *project* into *subprojects* through *subcontracting*. In a computer system, we classify data objects into *types*. This is a *granulation* [27]. Therefore, *granulation* is an essential feature in the design of programming languages. It provides predefined *basic types* and *user-defined types* so that the language can extend its application domain to any desired area.

Programming language likes *FORTRAN IV* have only *basic types* and do not allow users to define their own types. Therefore, its application domain is so limited. However, the *Ada* and the *C++* programming languages do provide *user-defined* type capability; and their application domain is not limited [26]. Today, these two languages are considered general purpose programming languages. Hence, *user-defined* types are vital for this complex and chaotic world in computing.

From an engineering viewpoint, we usually subdivide a complicated problem or system into several sub-problems or subsystems respectively; this method reduces the difficulty of solving the problem. In the next section, we will address subprograms and communications between program units for synchronization of the world.

III. MODULARIZATION AND COMMUNICATION IN THE CHAOTIC WORLD

Modularity partitions a complicated problem into sub-problems, and we implement them as *sub-programs*. A *sub-program* is a unit of a program declared out of line and invoked via calls. The purposes of a *subprogram* are many folds:

(1) Result of *modular design* of program,
(2) *Factoring of common logic* that occurs several places in a program,
(3) *Parameterized* calls allow operating on different objects at different times, and
(4) Simplifying and easing the complexity of the problem.

In general, there two distinct subprogram types exist in commonly used programming languages.

(1) **Functions**. It returns values of a designated type. These are used anywhere an expression is accepted.
(2) **Procedures**. It represents computational segments, and its results may be passed back via parameters or side effects. These are used wherever statements are permitted.

Communication between the main program and subprograms or from one subprogram to another occurs via *parameter passing*. Each programming language defines its own *parameter passing* mechanisms. There are five *parameter-passing* mechanisms in current commonly used programming languages such as parameter passing by *value*, by *reference*, by *name*, by *value-result*, and by *copy rules* that including *copy- in*, *copy-out*, and *copy-in out* rules [15, 16].

Among these five parameter-passing mechanisms, *parameter passing by value*, *parameter passing by reference*, and *parameter passing by name* can be defined mathematically below: For a parameter pass, in evaluating a variable *name* to get a *value* by finding the *location* associated with the *name* and extracting the *value* from the *location*. Let **names**, **locations**, and **values** be three sets, we define two mappings:

ρ : **names** $\rightarrow$ **locations**, and

σ : **locations** $\rightarrow$ **values**

Differences in the parameter passing mechanism are defined by when ρ and σ are applied.

(1) Parameter passing by **value**
ρ and σ both applied at point of call, argument completely evaluated at point of call.

(2) Parameter passing by **reference** (location)
Location is determined at the point of call and location is bound as the value of parameter. ρ is applied at point of call and σ is applied with every reference to the parameter.

(3) Parameter passing by **name**
At time of call, neither ρ nor the σ is applied. ρ and σ are applied with every reference to the parameter.

These three parameter-passing mechanisms formed hierarchical structures; the diagram is given in Fig.3.1. For parameter *passing by value*, both ρ and σ are applied at *calling time*; for parameter *passing by reference*, ρ is applied at *calling time* and σ is applied at reference time; for parameter *passing by name* both ρ and σ are applied with every reference to the parameter.

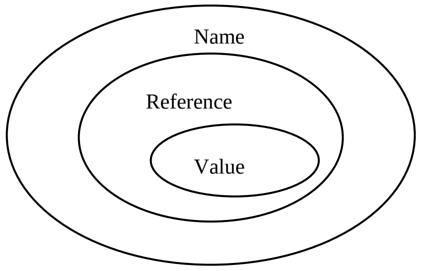

Fig. 3.1 Hierarchical structures of parameter passing

The *ALGOL*, *C*, *C++*, and *Pascal* languages provide parameter **passing by value** mechanism; and it is a convenient and effective method for enforcing write protection. The *ALGOL*, *C*, *C++*, and *Pascal* languages also implement parameter **passing by reference**. This eliminates duplication of memory. But, there are disadvantages in the parameter passing by **reference**. First, it will likely be slower because one additional level memory addressing is needed compared to parameter passing by **value.** Second, if only one-way communication to the called subprogram is required, unexpected and erroneous changes may occur in the actual parameter. Finally, parameter passing by **reference** can create **aliases**, which are harmful to *readability* and *reliability*. They also make program verification difficult.

The *ALGOL* programming language by default provides parameter passing by **name**. When implementing parameter passing by **name**, the system will create a *run-time* subprogram to evaluate the expression in the calling unit of the program and return the result to the called unit of the program. Therefore, it requires some additional overhead to implement such a *run-time subprogram*. In the *ALGOL* programming language, if one wants parameter passing by **value**, he or she must declare with the word "**value**" for that variable in the actual parameter. *ALGOL* treats parameter passing by **reference**, as a special case of parameter passing by **name**; therefore, the programmer does not need to specify anything. We can define a hierarchical structure for parameter passing by ***value*, reference**, and **name** in the *ALGOL* language.

In the programming language *PL*/1, for actual parameters with a single variable, *PL*/1 uses parameter passing by **value**. However, for a constant or an expression as an argument in the calling statement, *PL*/1 refers to a dummy formal parameter in the called subprogram and it implements a default value or parameter passing by **value.**

In most *FORTRAN* implementations before *FORTRAN* 77, parameters were passed by reference. In later implementations parameter passing by **value-result** has been used commonly.

For the *Ada* language, parameter passing has three modes: mode **in**, mode **out**, and mode **in out**. These are different from parameter passing by value and by reference [3, 4, 17, 20]

Mode **in**

- This is the default mode (i.e., **in** may be omitted).
- The actual parameter is **copied** into a local variable. The actual parameter must have a defined value at the point of call.
- The actual parameter may be an expression of compatible type.

Mode **out**

- The result is **copied** into the actual parameter upon exit.
- The actual parameter may be a variable of compatible type.
- The actual parameter need not have a value upon entry.

Mode **in out**

- The value of the actual parameter is **copied** into a local variable upon entry.
- The value of local parameter is **copied** into the actual parameter upon exist.
- The actual parameter must be a variable with a defined value upon entry.

These parameter-passing mechanisms are serving communication facilities between main program and its subprograms or between one subprogram and another. The purpose of engineering computing is to solve problems in the complicated world by means of *granulation* [27, 28], *organization*, and *causation*. *Granulation* subdivides the problem into a set of more manageable sub-problems. *Granulation* is an effective tool to modularize the original problem and to write it into subprograms. From a programmer viewpoint, communication between the main program and subprograms and the communication from one subprogram to another is the *causation* within the program. The structural design and the logical flow reflect the *organization* of the problem. Every parameter passing mechanism designed in current programming languages has its own purpose and application domain. For the *granulation computing*, we need to provide all of these features for selection so that we can solve problems most flexibly.

This paper emphasizes using features of programming languages for engineering computation. It is worth noting that the *Ada* programming language is designed for *embedded systems*, *safety-critical software*, and *large projects* that require high *portability*, *reliability*, and *maintainability*. For example, over 99 percent of the aviation software in the *Boeing* 777 airplane uses the *Ada* language [1]. Not surprisingly, the *Ada* language was the first object-oriented design programming language to be accepted as an International Standard.

Today, we design software systems to meet complex application requirements. Almost all the activities in human society, the biological world, physical systems, and engineering projects are *concurrent* or *parallel*; and purely *sequential* activities are special cases. Therefore, *concurrency* reflects the nature of designing software projects. In the next section, we will address the *multitasking* features in the *Ada* programming language [6, 9].

IV. PARALLEL OR CONCURRENCY IS THE NATURE OF THE WORLD

Among all commonly used programming languages, the *Ada* language has the most complete and best features for **multitasking. Multitasking** permits a programmer to partition a big job into many parallel **tasks** [8, 20]. Other programming languages like the *C* and *C++* programming languages can only apply some predefined functions. Thus, they are lack of flexibility and limit their applicability. For engineering applications, we should use these features and include them in the design of programming languages.

A **task** is a unit of computation that can be scheduled independently and in parallel with other such units. An ordinary *Ada* program can be thought of as a **single task**; in fact, it would be called the **main task**. Other tasks must be declared in the **main task** (as **subtasks**) or be defined in a **package** [3, 5, 20, 21]. Several independent *tasks* are often to be executed *simultaneously* in an application. *Ada tasks* can be executed in true *parallelism* or with apparent concurrency simulated by *interleaved execution*. *Ada tasks* can be assigned relative **priorities** and the underlying operating system can schedule them accordingly. A *task* is terminated when its execution ends. A *task* can be declared in *packages*, *subprograms*, *blocks*, or *other tasks*. *All tasks* or *sub-tasks* must terminate before the declaring *subprogram*, *block*, or *task* can be terminated.

A task may want to communicate with other tasks. Because the execution speed of tasks cannot be guaranteed, a method for synchronization is needed. To do this, the *Ada* language requires the user to declare **entry** and **accept** statements in two respective tasks engaged in communication. This mechanism provides for *task* interaction and is called a **rendezvous** in the *Ada* language [21].

The *Ada* language also gives an optional scheduling called a **priority** that is associated with a given *task*. A **priority** expresses relative urgency of the *task* execution. An expression of a **priority** is an integer in a given defined range. A numerically smaller value for **priority** indicates lower level of urgency. The **priority** of a *task*, if defined, must be static. If two *tasks* with no **priorities** or two *tasks* of equal priority exist, they will be scheduled in an *arbitrary order*. If two tasks of different priorities are both eligible for execution, they could sensibly be executed on the same processor. A lower priority task cannot execute while a higher priority *task* waits. The *Ada* language forbids time-sliced execution scheduling for *tasks* with explicitly specified priorities. If two *tasks* of prescribed priorities are engaged in a **rendezvous**, the **rendezvous** is executed with the higher of the two priorities. If only one *task* has a defined priority, the **rendezvous** is executed at least at that priority.

A *task* may **delay** its own execution or put itself to *sleep* and *not use processing resources* while waiting for an event to occur by a *delay statement*. The **delay** statement is employed for this purpose. Zero or negative values have no effect. The *smallest delay time* is **20 milliseconds** or **0.020 seconds**. The m*aximum delay* duration is up to **86400 seconds** or **24 hours**. The duration only specifies minimum delay; the task may be executed any time thereafter, if the processor is available at that time.

The *Ada* language also provides a **select statement**. There are three forms of select statements. Selective wait, conditional **entry call**, and **timed entry** call. A **selective wait** may include (1) a terminate alternative, (2) one or more **delay alternatives**, or (3) an **else** part, but only one of these possibilities is legal. A task may designate a family (an array) of *entries* by a single name. They can be declared as:

```
Entry request (0..10) (reqcode: integer);
Entry alarm (level);           -- where type level
                               -- must be discrete
```

An accept statement may name an indexed entry

```
Accept request ( 0) (reqcode: integer) do …
Accept request (1) (reqcode: integer) do …
Accept alarm (level);
```

An entry family allows an accepting task to select entry calls to the same function deterministically.

Today's compiler technology adequately supports all *Ada* features. In a real-time system, the response time for *multitasking features* may seem not fast enough. Therefore, speed is an important factor in choosing an *Ada compiler* for general and real-time system applications. Among all commonly used programming languages, the *Ada* language is the unique one that provides *multitasking* features at the programming level, and it is very important useful feature for *modeling and simulating* of *real-time and concurrent events* in programming.

The goals of computing are *reliability*, *efficiency*, *accuracy*, and *ease of use*. From the programming point of view, to provide reliable computation is to prevent or eliminate *overflow*, *underflow*, and other *unexpected conditions* so that a program can be executed safely, completely, and efficiently. *Efficient computation* requires an effective computational algorithm for the given problem using proper programming language features for that computation. For *accurate computation*, one should consider problem solving capability, accuracy features, and parallel computation abilities in a given programming language. For *ease of use*, the software engineer should put himself in the situation of the user. A software engineer should remember that users have a job to be done, and they want the computer system to do the job with a minimum of effort. In the next section, we will discuss issues of **accuracy** and **efficiency** of the *Ada* language in *numerical computation capability* [15].

V. ACCURACY AND EFFICIENCY ARE REQUIRED BY THIS SOPHICIFICATE WORLD

The area of *numerical computation* is the backbone of computer science and all engineering disciplines. *Numerical computation* is critical to real world engineering applications. For example, on November 10, 1999, the U.S. National Aeronautics and Space Administration (*NASA*) reported that the *Mars Climate Orbiter Team* found:

"*The 'root cause' of the loss of the spacecraft was the failed translation of English units into metric units in a segment of ground-based, navigation-related mission software as NASA previously announced* [11]."

This example indicates that numerical computation and software design are crucial tasks in an engineering project. The goal of *numerical computation* is to reach to a sufficient level of accuracy for a particular application. Designing software for efficient computation is another challenge.

For *engineering applications*, we need to deal with many of *numerical computations*. Among most commonly used programming languages, the *Ada* language has the best numerical computation capability. From the **precision** aspect, the *Ada* language allows a user to define his own accuracy requirement. This section will address the *Ada* language numerical computation capability [15]. To deal this, we should consider the following four criteria: problem solving capability, accuracy of computation, execution time for solving problems, and the capability of parallelism

(1) Problem solving capability: The *Ada* language provides user-defined types and separate compilation. The former supports programmers solving a wide range of engineering problems and the latter permits development of large software systems. The *Ada* language provides data abstraction and *exception handling* that support information hiding and encapsulation for writing a reliable program. In the real world, many engineering projects consist of concurrent or parallel activities in their physical entities. To properly simulate these systems with a programming language, that language must provide for logical concurrency regardless of how the program is actually executed. *Ada multitasking* meets this requirement [13, 22]. In fact, multiprocessor computer systems are now available, thus simulating a truly parallel system becomes possible.

(2) Precision and accuracy: The *Ada* language's real number types are subdivided into *float-point types* and *fixed-point types*. *Float-point type* have values are numbers with the format, $\pm .dd..d \times 10^{\pm dd}$. *Fixed-point types* have values with the formats $\pm dd.ddd$, $\pm dddd.0$ or $\pm 0.00ddd$ [4, 8, 20].

For the *float-point number types*, **model numbers** other than zero, the numbers that can be represented exactly by a given computer, are of the form:

sign × *mantissa* × (*radix* ×× *exponent*)

In this form, *sign* is either +1 or -1; *mantissa* is expressed in a number base given by *radix* and *exponent* is an integer. The *Ada* language allows the user to specify the number of significant decimal digits needed. A floating-point type declaration with or without the optional range constraint is shown:

type *T* **is digit** *D* [**range** *L* .. *R*];

In addition, most *Ada* compilers provide the types *long_float* and *long_long_float* (used in package standard) and *f_float*, *d_float, g_float*, and *h_float* (used in package system) [22]. The size and the precision of each of the *Ada* floating-point types are given as follows:

Type	Size(bits)	Precision (digits)
f_float	32	6
d_float	64	9
g_float	64	15
h_float	128	33

The goal of computation is accuracy. Higher accuracy will provide more reliability in the real-time environment. Sometimes, a single precision or a double precision of floating point numbers in *FORTRAN* 77 [13] is not enough for solving some critical problems. In the *Ada* language one may use the floating point number type: *long_long_float* (*h_float*) by declaring digit 33 to use 128 bits for floating point numbers provided by Vax *Ada* [18] to provide a precision of 33 decimal digits accuracy, and the range of exponent is about from 10^{-134} to 10^{+134} or –448 to +448 of base 2 [4, 8, 19, 20] for the range. The author has employed this special accuracy feature in the computation of *hypergeometric distribution function* [23, 24].

For the *fixed-point types*, the **model numbers** are in this form:

sign × *mantissa* × *small*

The sign is either +1 or –1; *mantissa* is a positive integer; small is a certain positive real number. *Model numbers* are defined by a fixed-point constraint, the number small is chosen as the largest power of two that is not greater than the **delta** of a fixed accuracy definition. The *Ada* language permits the user to determine a possible range and an error bound which is called **delta** for computational needs. Examples are the follows:

Overhead has a **delta** of 0.01;

Overhead has a **range** −10*E*5 .. 1.0*E*5;

These indicate *small* is 0.0078125 which is 2^{-7} and model numbers are –12800000 × *small* to +12800000 × *small*. The predetermined range provides a reliable programming environment. The user assigned error bound delta guarantees an accurate computation. These **floating-point number** and **fixed-point number types** not only provide good features for real-time critical computations, but also give extra reliability and accuracy for general numerical computations.

(3) The *Ada* for parallel computation: The author has used *exception handlings* and *tasks* [6, 21] for computation of a division of a product of factorials and another product of factorials in the computation of the *hypergeometric distribution function* [24, 25]. *Exception handling* is used to prevent an *overflow* or *underflow* of multiplications and divisions, respectively. The tasks are used to compute the numerator and denominator concurrently. In addition, *tasks* and *exception handling* working together can minimize the number of divisions and maximize the number of integer multiplications in both of the numerator and denominator, reduce round off errors, and obtain the maximum accuracy. In the actual implementation, we used **three tasks**: *task one* and *task two* are employed to perform the multiplications in the numerator and denominator parallels and *task three* is used to perform a division. When both products in the numerator and denominator have reached a maximum before an overflow occurs, both *task one* and *task two* stop temporarily and invoke *task three* to perform a division of the products that have been obtained in the numerator and denominator before an overflow occurs. After *task three* completes its job, *task one* and *task two* resume their computation and repeat this procedure until the final result is obtained. These **tasks** work together and guarantee that the result of this computation will be the most accurate and the time for the computation is reasonable. The author has performed these computations on a single processor machine, so the parallelism is a logical parallelism. If one has a multiprocessor machine, he can perform an actual parallelism, tasking and exception *handling* can easily be employed in the computation of the *hypergeometric distribution function* and some computation results and required time for this problem are given in [21], along with those for the computation of the *multinomial distribution function,* multivariate *hypergeometric distribution function*, and other comparable functions. We conclude here that it is not possible to carry out the computation without using these *Ada* special features.

(4) Execution time: In the 1990s, compiler technology was inadequate to support many *Ada* features. In a real-time system, the response time for *multitasking* features was seemed not fast enough. Therefore, speed was an important criterion for choosing an *Ada* compiler for *real-time* applications. However, the second generation of *Ada* compilers has doubled the speed of the first generation compilers. Today, compilers are fast enough to support all *Ada* features. Currently, *Ada* compilers are available for supercomputers, mainframe computers, minicomputers, and personal computers at reasonable prices.

In running an application, a program *crash* is a disaster. When the programmer designs his program, he must consider all such unexpected situations and find a means to prevent from happening. If it is not possible to prevent a *crash* from occurring, the programmer should provide some mechanisms to handle it, to eliminate it, or to minimize its damage. In the next section, we will address *exception-handling* features for these purposes.

VI. EXCEPTION HANDLING IS THE SAFE GUARD IN THE DANGEROUS WORLD

An **exception** is an *out-of-the-ordinary* condition, usually representing a *fault state* that can cause a program crash. The *Ada* language provides for *detection* and *handling* of such conditions [4, 8, 23]. It is raised implicitly from an operation of integer overflow during the evaluation of an expression, or assigning a negative value to a type with positive data item. It is also raised explicitly as a result of checking when a determinant is found to be zero during matrix inversion or a *stack* is found to be empty during "*pop*" operation. It is not always possible or practical to avoid all exceptions. Allowing exceptions to occur without providing a means to handle the condition could lead to an *unreliable program* or a *crash*, however. Therefore, *exception handling* is very important in designing a programming language [17, 23]. The *Ada* language provides five predefined exceptions. They are:

Constraint_error

- It possibly the most frequently used exception in *Ada* programs.
- Constraint_error is raised when a range constraint is violated during assignment or a discriminant value is altered for a constrained type.

Numeric_error
- When an arithmetic operation cannot deliver the correct result.
- Overflow or underflow condition occurs.

Storage_error
- It is raised if out of memory during the *elaboration* of a data object.
- During a subprogram execution, creation of a *new-access type* object,

Tasking_error
- It is raised during *inter-tasking communication*.

Program_error
- A program attempts to enter an unelaborated procedure, e.g. a *forward* is not declared.

The *Ada* language encourages its user to define and raise his exceptions in order to meet the specific purposes needed. The language allows the *exception handling* mechanism to be invoked for unusual conditions that are detected by the user's program. An **exception handler** is a segment of subprogram or block that is entered when the exception is raised. It is declared at the end of a block or subprogram body. If an *exception* is raised and there is no handler for it in the unit, then the execution of the unit is abandoned and the *exception* is *propagated dynamically*, as follows:

- If the unit is a block, the *exception* is raised at the end of the block in the containing unit.
- If the unit is a *subprogram*, the *exception* is raised at the point of call rather than the statically surrounding units; hence the term dynamic is applied to the propagation rule.
- If the unit is the *main program*, program execution is terminated with an appropriate (and nasty) diagnostic message from the run time support environment.

The *Ada* language strongly encourages its user to define and use his own exceptions even if they manifest in the form of predefined exceptions at the outset, rather than depend on predefined ones; it is generally difficult to be sure exactly what caused the original exception. The user should not depend on proper values for **out** or **in out** parameters from units that neither **propagate** an *exception* nor assume a predefined exception occurs at some known point even if there is only one point where it could occur. The following is an example that shows the *exception*, *handler*, and *propagation* working together for the computation of factorial function.

```
Function Power (n, k: natural) return natural is
    Function Power (n, k: natural) return natural is
        begin                                         -- inner
            if k < 1 then
                return 1;
            else
                return n * Power(n, k-1);
        end if;
    end Power;                                    -- inner
begin                                                 -- outer
    return Power(n, k);
    Exception
        when numeric_error =>
    return natural'last;
end Power;                                            -- outer
```

The advantage of this segment of code is that when an overflow condition occurs, the inner **Power** function will exit. Once the outer function returns the **natural'last**, it is not possible to get back to the exact point of the exception.

If the function had been written as on the next page, each execution of the function would encounter an exception when an overflow occurs. An undesired example is given as follows for comparison. This function will continuously raise exceptions when the first *overflow* is detected and at the end of all the recursive calls.

```
Function Power (n, k: natural) return natural is
    begin
        if k < 1 then
            return 1;
        else
        return n * Power(n, k-1);
        end if;
        Exception
            when numeric_error =>
                return natural'last;
    end Power;
```

Designing a programming language for application to all human activities and their needs is not an easy task. However, the language we want for engineering computation must be a reliable one. The *Ada* language is a language that provides complete **exception-handling** facilities. For a program to be considered *reliable*, it must be operate sensibly even when presented with improper input data as well as a software or hardware malfunction. A wide range of issues pertains to the design and implementation of software, which will operate successfully within specifications during specified time periods. Of cause a greater precaution is to avoid possible errors in the program itself. To do this, we must take account of such possible errors in the program design. A truly reliable program must monitor itself during execution and take some appropriate actions when a computation is entering an unexpected state. This is the purpose of an **exception handler**.

VII. Conclusion

In this paper we have discussed the goal of engineering computing as *accuracy*, *efficiency*, *reliability*, and *ease of use*. To accomplish the goals of engineering computation is not easy; it involves human intelligence, knowledge of variety of programming languages, and various programming facilities. In this paper, we have examined many programming language features that are critically important for engineering applications. Some of these features such as *subprograms*, *user-defined types*, and *basic numerical computation facilities* are provided by most commonly used languages like *FORTRAN* 90, the *C*, the *C++*, *Pascal*, and *Ada* languages. This paper has highlighted the use of the *Ada* programming language's *strong typed feature*, *predefined exception handling*, *user defined exception handling*, and *user-defined types for developing reliable programs*. All of these good features in the *Ada* programming languages are unique among all commonly used programming languages. In particular, the use of the *Ada* language's **delta**, **digits**, and **model numbers** for designing engineering projects, which require accurate critical numerical computation, are very important. For the *Ada* language, most of their compilers allow users to have a 128-*bit floating-point number* or 33 *significant digits* for numerical computation. Programmers may specify required accuracies through **digits** and **delta clauses** for *floating-point numbers* and **fixed-point numbers**, respectively. In addition, *Ada's exception handling* can prevent *overflow* or *underflow* during the execution of programs and *multitasking* can perform *parallel computations*. Therefore, from the software reliability point of view, the *Ada* language is better than *FORTRAN*, the *C*, the *C++* and languages in *numerical computation* for engineering applications.

Today, most engineers have only *FORTRAN* or the *C/C++* programming languages for computing. Perhaps some of them have used **Mathematica**, **Mathlab**, or **Maple** for computation. All these mathematical packages have one or more of following drawbacks. Each mathematical software package has its own input and output formats; these formats might not compatible with an engineering project. Each of these software packages has its predefined accuracy, which might not meet the needs of engineering projects. All of these mathematical packages are designed for classroom teaching or laboratory research computations; *efficiency* is not critically important for these purposes. Moreover, these mathematical packages are not for *real-time* or *embedded systems* applications; they lack *exception-handling* capabilities. Therefore, engineers need to learn how to build their own computational capabilities.

To do this, each engineering student needs to take at least two courses in computing, one is to learn the basic computation capabilities, a *FORTRAN* or the *C/C++* course will serve for this purpose, the other is to study how to build *efficient*, *accurate*, *reliable*, and *ease of use* software systems to satisfy all engineering domain needs. However, the instructor must have knowledge about engineering experiences in real world, and backgrounds in inter-disciplinary applications in order to qualify for this purpose and lead students to design, implement, and handle engineering projects for the challenges of the sophisticated word.

References

1. Ada information clearance house, the web site: www.adaic.org/atwork/boeing.html
2. *The Aviation Book, A visual encyclopedia of the history of aircraft*, www. ChronicaleBooks.com.
3. Barnes, J. G. P. *Programming in Ada*, Third edition, Addison-Wesley Publishing, Reading Massachusetts, 1989.
4. Barnes, J. *Programming in Ada 95*, Addison-Wesley Publishing, Reading Massachusetts, 1995.
5. Boeing 777, globalsecurity.org/military/systems/ aircraft/b777.html.
6. Booch, G. *Software engineering with Ada*, Second edition, Benjamin/Cummings Publishing, Reading Massachusetts, 1987.
7. *China Daily*, CHINAdaily.com.cn, October 28, 2006.
8. Demillo, R. and Rice, J. eds, *Studies in Computer Science: In Honor of Samuel D. Conte*, Plenum Publishing, 1994.
9. Habermann, A. N. and Perry D. E. *Ada for Experienced Programmers*, Addison-Wesley Publishing, Reading, Massachusetts, 1983.
10. Joint Task Force for Computing Curricula 2004, *Overview Report.* ACM and IEEE-CS. 2004.
11. Mas project http://mars.jpl.nasa.
12. Myers, G. J., *Software Reliability*, John Wiley & Sons, 1976.
13. Parnas, D. L. Is *Ada* Too Big?" (letter) *Communications of ACM* 29, pp. 1155-1155, 1984.
14. Pfleeger, S. L. *Software Engineering, the Production of Quality Software*, 2nd edition, Macmillam, 1991.
15. Pratt, T. W., Zelkowitz, M.V. *Programming Languages, Design and implementation*, third edition, Prentice-Hall, Englewood Cliffs, New Jersey, 1996.
16. Sebesta, R. W. *Concepts of Programming Languages*, sixth edition, Addison-Wesley Publishing, Reading Massachusetts, 2003.
17. Smedema, C. H., et al., *The Programming languages Pascal, Modula, Chill*, and *Ada*, Prentice-Hall, Englewood Cliffs, New Jersey,1983.

18. Struble, G. *Assembler Language Programming*, The IBM System/370 Family, Addison-Wesley Publishing, Reading, Massachusetts 1984.
19. *Vax Ada Language Reference Manual*, Digital Equipment Corporation, Maynard, Mass., 1985.
20. Watt, D. A., et al. *Ada Language and Methodology*, Prentice-Hall, Englewood Cliffs, New Jersey, 1987.
21. Welsh, J. and Lister, A. A Comparative Study of Task Communication in *Ada*, *Software Practice and Experience*. 11, pp. 257-290, 1981.
22. Wichmann, B. A. Is *Ada* Too Big? A Designer Answers the Critics, *Communications of ACM*, 29, pp. 1155-1156, 1984.
23. Wu, T. Built-in reliability in *the Ada* programming language, *Proc. of IEEE 1990 National Aerospace and Electronics Conference*, pp. 599-602, 1990.
24. Wu, T. An Accurate Computation of the Hypergeometric Distribution Function, *ACM Transactions on Mathematical Software*, V.19, No. 1, pp. 33-43, 1993.
25. Wu, T. *Ada* programming language for Numerical Computation, *Proc. of IEEE 1995 National Aerospace and Electronics Conference*, pp. 853-859, 1995
26. Wu, T. Some tips in computer science, a talk given at University of Illinois at Springfield, Dec. 2, 2004.
27. Zadeh, L. A. Fuzzy sets and information granularity, In M. M. Gupta, P. K. Ragade, R. R.Yager, eds, *Advances in Fuzzy Set Theory and Applications*, North Holland, Amsterdam, pp. 3-18, 1979.
28. Zadeh, L. A. Towards a theory of fuzzy information granulation and its centrality in human reasoning and fuzzy logic, *Fuzzy Sets and Systems* V. 19, pp.111-117, 1997.