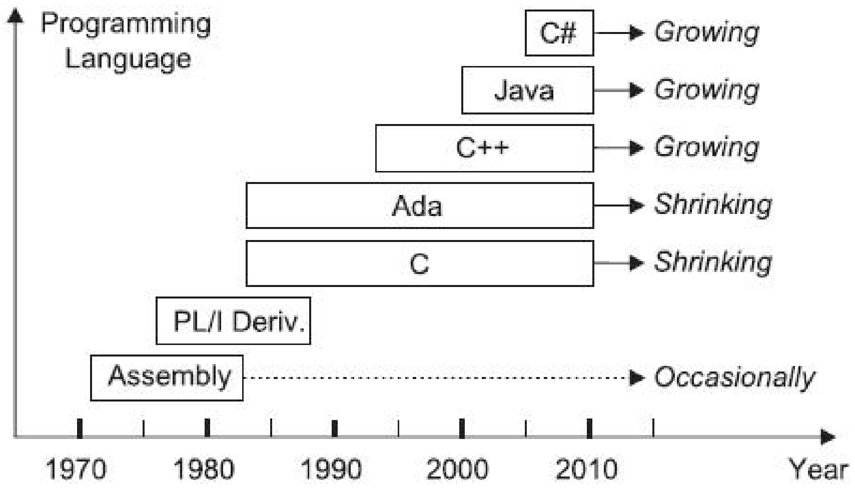
PROGRAMMING LANGUAGES FOR REAL - TIME SYSTEMS

4.1 CODING OF REAL - TIME SOFTWARE

Misuse of the programming language can be the single greatest source of performance deterioration and missed deadlines in real - time systems.

Moreover, when using object - oriented languages in real - time systems, such performance problems can be more difficult to analyse and control. Nonetheless, object - oriented languages are steadily displacing procedural languages as the language of choice in real - time embedded systems development. Figure below depicts the *mainstream* use of programming languages



Fitness of a Programming Language for Real - Time Applications

A programming language represents the nexus of design and structure. Software depends on tools to compile, generate binary code, link, and create binary objects should take proportionally less time than the requirements engineering and design efforts.

The main tool in the code generation process is the language compiler. Real - time systems are currently being built with a variety of programming languages including various dialects of C, C + + , C#, Java, Ada, assembly language, and even Fortran or Visual Basic.

Then " What is the fitness of a programming language for real - time applications and what metrics could be used to measure or at least estimate such fitness? " To address this multidimensional question consider, for instance, the five criteria must follow:

C1. Economy of Execution (How fast ?)

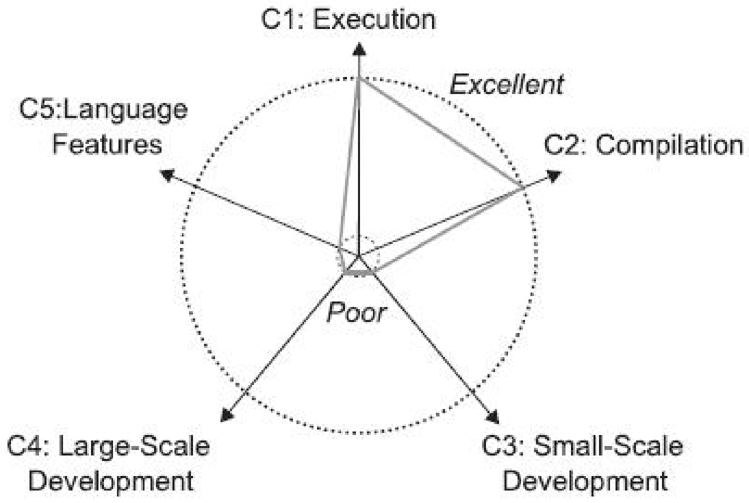
C2. Economy of Compilation (single or multiple file)

C3. Economy of Small - Scale Development . How hard to design)?

C4. Economy of Large - Scale Development . How hard for a team of programmers?)

C5. Economy of Language Features. How hard to learn and programming?)

language?
Every programming language offers undoubtedly its own strengths and weaknesses with respect to real - time systems, and these qualitative criteria, C1 – C5, can be used to calibrate the features of a particular language for oranges - to -oranges comparison within a given application. The criteria can be illustrated with a pentacle diagram shown in Figure 4.2 .



The compile - time prediction of execution performance directly supports a schedulability analysis. In the design of special real – time programming languages, the emphasis is on eliminating those constructs that render the language nominalizable, for example, unbounded recursion and unbounded while loops. Most so- called " real- time languages" strive to eliminate all of these.

Coding Standards for Real - Time Software

Coding standards are different from language standards.
A language standard embodies the syntactic rules of the C ++ programming language. A source program violating any of those rules will be rejected by the compiler. A coding standard, on the other hand, is a set of stylistic conventions or " best practices. " Violating these conventions will not lead to compiler rejection. In another sense, compliance with language standards is mandatory, while compliance with coding standards is, at least in principle, voluntary.
Adhering to language standards fosters portability across different compilers, and, hence, hardware environments. Complying with coding standards, on the other hand, will not foster portability, but rather in many cases, readability, maintainability, and reusability. Some practitioners even contend that the use of strict coding standards can increase software reliability. Coding standards may also be used to foster improved performance by encouraging or mandating the use of certain language constructs that are known to generate code that is more efficient.

Coding standards typically involve standardizing some or all of the following elements of programming language use:
• Header format
• Frequency, length, and style of comments
• Naming of classes, data, fi les, methods, procedures, variables, and so forth
• Formatting of program source code, including use of white space and indentation
• Size limitations on code units, including maximum and minimum number of code lines, and number of methods used
• Rules about the choice of language construct to be used; for example, when to use case statements instead of nested if-then-else statements.

There exist many different standards for coding that are either language independent or language specific. Coding standards can be companywide, teamwide, user - group specific (e.g., the GNU software group has standards for C and C ++), or customers can require conformance to a certain standard of their own. Furthermore, other standards have come into the public domain. One example is the Hungarian notation standard (Petzold, 1999), named in honor of Charles Simonyi, who is credited with first promulgating its use.
Hungarian notation is a public - domain standard intended to be used with object - oriented languages, particularly C ++ . The standard uses a purposeful naming scheme to embed type information about the objects, methods, attributes, and variables in the name. Because the standard essentially provides a set of rules about naming various identifiers, it can be and has been used with other languages, such as Ada, Java, and even C, as well. Another example is in Java, which, by convention, uses all uppercase for constants such as PI and E. Moreover, some classes use a trailing underscore to distinguish an attribute like x_ from a method like x( ) .
A general problem with style standards, like the Hungarian notation, is that they can lead to mangled variable names, and that they direct the programmer ' s focus on how to name in " Hungarian " rather than choosing a meaningful name of the variable for its use in code. In other words, the desire to conform to the standard may not always result in a particularly meaningful variable name. Another problem is that the very strength of a coding standard can also be its undoing. For example, in Hungarian notation, what if the type information embedded in the object name is, in fact, wrong? There is no way for any compiler to recognize this mistake. There are commercial rules wizards, reminiscent of the C language checking tool, lint, which can be tuned to enforce coding standards, but they must be programmed to work in conjunction with the compiler. Moreover, they can miss certain inconsistencies, leading the developers to a sense of false confidence.
Finally, adoption of coding standards is not recommended mid - project. It is easier and more motivating to start conforming from the beginning than to be

required to change an existing style to comply. The decision to use a specific coding standard is an organizational one that requires significant forethought and open discussion.

ASSEMBLY LANGUAGE

In the mid - to late 1970s, when the first high - level languages became available for microprocessors but limited role in real - time programming.

Reasons --

- assembly language does have a particular advantage for use in real - time programming; it provides the most direct control of the computer hardware over high - level languages.
- This advantage has extended the use of assembly language in real - time systems, despite the fact that assembly language is unstructured and has very limited abstraction properties.
- Coding in assembly language is, in general, time - consuming to learn, tedious, and error prone.
- Finally, the resulting code is not easily ported across different processors, and hence the use of assembly language in embedded real - time systems — or in any professional system — is to be discouraged.
- But with significant improvements in optimizing compilers, the compiler should be able to generate very efficient machine - language code in terms of execution speed and memory usage.
- Furthermore, you will find assembly - language code in many legacy real – time applications, and even today you can still occasionally encounter situations where small portions of a real - time system need to be written using assembly language.
- Assembly languages, however, have poor economies of small - and large - scale development and of language features. Hence, assembly language programming should be limited to use in very tight timing situations or in controlling hardware features that are not supported by the compiler.

The continuing role of assembly language in this decade is summarized below:

- For certain kinds of code, such as interrupt handlers and for device drivers for unique hardware where the “ intellectual distance ” between the hardware and software needs to be minimized.
- For situations where predictable performance for the code is extremely difficult or impossible to obtain because of undesirable programming - language – compiler interactions.
- For effectively using all architectural features of a CPU, for instance, parallel adders and multipliers.
- For writing code with minimum execution time achievable for time – critical applications, such as sophisticated signal - processing algorithms with high sampling rates.

- For writing the entire software for custom - designed CPUs with a small instruction set (see Section 2.5.3) — if no high - level language support is available.
- For debugging hard problems below the level of high - level language code and tracing the stream of fetched instructions by a logic analyser.
- For teaching and learning computer architectures and internal operation of processors.

To deal with these special situations, the software developer will usually write a shell of the program in a high - level language and compile the code to an intermediate assembly representation, which is then fine - tuned manually to obtain the desired effect. Some languages, such as Ada, provide a way for assembly code to be placed inline with the high - level - language code. In any case, the use of assembly language within a real - time system must be done reluctantly and with extreme caution.

PROCEDURAL LANGUAGES

Procedural languages such as Ada, C, Fortran, and Visual Basic, are those in which the action of the program is defi ned by a set of operations executed in sequence. These languages are characterized by facilities that allow for instructions to be grouped together into procedures or modules. Appropriate structuring of the procedures allows for achievement of desirable properties of the software, for example, modularity, reliability, and reusability.
There are several programming - language features standing out in procedural languages that are of interest in real - time systems, particularly:

- Modularity
- Strong typing
- Abstract data typing
- Versatile parameter passing mechanisms
- Dynamic memory allocation facilities
- Exception handling

These language features, to be discussed shortly, help promote the desirable properties of software design and best real - time implementation practices.

Modularity and Typing Issues

- Procedural languages that are amenable to the principle of information hiding tend to promote the construction of high - integrity real - time systems.
- While C and Fortran both have mechanisms that can support information hiding (procedures and subroutines), other languages, such as Ada, tend to foster more *modular design* because of the requirement to have clearly defi ned inputs and outputs in the module parameter lists.
- In Ada, the notion of a package embodies the concept of Parnas information hiding (Parnas, 1972) exquisitely. The Ada package consists of a

specification and declarations that include its public or visible interface and its private or invisible elements. In addition, the package body, which has more externally invisible components, contains the working code of the package. Individual packages are separately compilable entities, which further enhances their application as black boxes. Furthermore, the C language provides for separately compiled modules and other features that promote a rigorous top – down design approach, which should lead to a solid modular design.

- While modular software is desirable for many reasons, there is a price to pay in the overhead associated with procedure calls and essential parameter passing. This adverse effect should be considered carefully when sizing modules.
- Typed languages require that each variable and constant be of a specific type and that each be declared as such before use.
- *Strongly typed* languages prohibit the mixing of different types in operations and assignments, and thus force the programmer to be exact about the way data are to be handled.
- Precise typing can prevent corruption of data through unwanted or unnecessary type conversion. Moreover, compiler type - checking is an important step to find errors at compile time, rather than at runtime, when they are more costly to repair. Hence, strongly typed languages are truly desirable for real - time systems.
- Generally, high - level languages provide integer and real types, along with Boolean, character, and string types. In some cases, *abstract data types* are supported, too. These allow programmers to defi ne their own types along with the associated operations. Use of abstract data types, however, may incur an execution - time penalty, as complicated internal representations are often needed to support the abstraction.
- Some languages are typed, but do not prohibit mixing of types in arithmetic operations. Since these languages generally perform mixed calculations using the type that has the *highest storage complexity*, they must promote all variables to that highest type. For example, in C, the following code fragment illustrates automatic promotion and demotion of variable types:

```
int x,y;
float a,b;
y=x*a+b;
```

Here the variable x will be promoted to a float (real) type and then multiplication and addition will take place in floating point. Afterward, the result will be truncated and stored in y as an integer. The negative performance impact is that hidden promotion and more time - consuming arithmetic instructions are generated, with no additional accuracy achieved. Accuracy can be lost due to the truncation, or worse, an integer overflow can occur if the real value is larger than the allowable integer value. Programs written in languages that are weakly typed need to be scrutinized for such effects. Fortunately, most C compilers can be tuned to catch type mismatches in function parameters, reventing unwanted type conversions.

Parameter Passing and Dynamic Memory Allocation

There are several methods of *parameter passing* , including the use of parameter lists and global variables. While each of these techniques has preferred uses, each has a different performance impact as well. Note that these parameter - passing mechanisms are also found in object - oriented programming languages.

The two most widely available parameter - passing methods are *call - by - value* and *call - by - reference* . In call - by - value parameter passing, the value of the actual parameter in the procedure call is copied into the procedure ' s formal parameter. Since the procedure manipulates the formal parameter only, the actual parameter is not altered. This technique is useful either when a test is being performed or the output is a function of the input parameters. For instance, in passing accelerometer readings from the 10 - ms cycle to the 40 – ms cycle, the raw data need not be returned to the calling routine in changed form.

When parameters are passed using call - by - value, they are copied onto a runtime stack, at additional execution - time cost.

In call - by - reference (or call - by - address), the address of the parameter is passed by the calling routine to the called procedure so that the corresponding memory content can be altered there. Execution of a procedure using call - by - reference can take longer than one using call - by - value, since in call - by - reference, indirect addressing mode instructions are needed for any operations involving the variables passed. However, in the case of passing large data structures, such as buffers between procedures, it is more desirable to use call - by - reference, since passing a pointer is more efficient than passing the data by byte.

- Parameter lists are likely to promote modular design because the interfaces between the modules are clearly defined. Clearly defined interfaces can reduce the potential of untraceable corruption of data by procedures using global access. However, both call - by - value and call - by - reference parameter – passing techniques can impact real - time performance when the lists are long, since interrupts are often disabled during parameter passing to preserve the integrity of the data passed. Moreover, call - by - reference may introduce subtle function side effects, depending on the compiler.

Before deciding on a specific set of rules concerning parameter passing for optimum performance, it is advisable to construct a set of test cases that exercise different alternatives. These test cases need to be rerun every time the compiler, hardware, or application changes in order to update the rules.

Global variables are variables that are within the scope of all code. " Within scope " usually means that references to these variables can be made with

minimal memory fetches to resolve the target address, and thus are faster than references to variables passed via parameter lists, which require additional memory references. For example, in many image - processing applications, global arrays are defi ned to represent entire images, hence allowing costly parameter passing to be avoided.
However, global variables are dangerous because references to them can be made by unauthorized code, potentially introducing faults that can be hard to isolate. Use of global variables also violates the principle of information hiding, making the code difficult to understand and maintain. Therefore, unnecessary and wanton use of global variables is to be avoided. Global parameter passing is only recommended when timing constraints so require, or if the use of parameter lists leads to obfuscated code. In any case, the use of global variables must be strictly coordinated and clearly documented.
The decision to use one method of parameter passing or the other may represent a trade - off between good software engineering practice and performance needs. For instance, often timing constraints force the use of global parameter passing in instances when parameter lists would have been preferred for clarity and maintainability.
Most programming languages provide recursion in that a procedure can either call itself or use itself in its construction. While recursion may be elegant and is sometimes necessary, its adverse impact on real - time performance must be considered. Procedure calls require the allocation of storage on the stack for the passing of parameters and for storage of local variables. The execution time needed for the allocation and deallocation, as well as for storing and retrieving those parameters and local variables, can be costly. In addition, recursion necessitates the use of a large number of expensive memory – and register - indirect instructions. Moreover, precautions need to be taken to ensure that the recursive routine will terminate, otherwise the runtime stack will eventually overfl ow. The use of recursion often makes it impossible to determine the exact size of runtime memory requirements. Thus, iterative techniques, such as while and for loops, must be used where performance and determinism are crucial or naturally in those languages that do not support recursion.
The ability to dynamically allocate memory is important in the construction and maintenance of many data structures needed in real - time systems. While *dynamic memory allocation* can be time - consuming, it is necessary, especially in the construction of interrupt handlers, memory managers, and the like.
Linked lists, trees, heaps, and other dynamic data structures can benefit from the clarity and economy introduced by dynamic memory allocation. Furthermore, in cases where just a pointer is used to pass a data structure, the overhead for dynamic allocation can be reasonable. When coding real – time systems, however, care should be taken to ensure that the compiler will always pass pointers to large data structures and not the data structures themselves. Languages that do not allow dynamic allocation of memory, for example, some

primitive high - level languages or assembly language require data structures of fixed size. While this may be faster, flexibility is sacrificed and memory requirements must be predetermined. Modern procedural languages, such as Ada, C, and Fortran 2003, have dynamic allocation facilities.

4.3.3 Exception Handling

Some programming languages provide facilities for dealing with errors or other anomalous conditions that may arise during program execution. These conditions include the obvious, such as fl oating - point overfl ow, square root of a negative argument, divide - by - zero, as well as possible user - defi ned ones. The ability to defi ne and handle exceptional conditions in the high - level language aids in the construction of interrupt handlers and other critical code used for real – time event processing. Moreover, poor handling of exceptions can degrade performance.

For instance, floating - point overflow errors can propagate bad data through an algorithm and instigate time - consuming error - recovery routines. In ANSI - C, the raise and signal facilities are provided for creating exception handlers. A signal is a type of software interrupt handler that is used to react to an exception indicated by the raise operation. Both are provided as function calls, which are typically implemented as macros.

The following prototype can be used as the front end for an exception handler to react to signal S .

```
void ( *signal (int S, void ( *func) (int)))(int);
```

When signal S is set, function func is invoked. This function represents the actual interrupt handler. In addition, we need a complementary prototype:

```
int raise (int S);
```

Here raise is used to invoke the task that reacts to signal S .

ANSI - C includes a number of predefi ned signals needed to handle anomalous conditions, such as overfl ow, memory access violations, and illegal instruction, but these signals can be replaced with user - defi ned ones. The following C code portrays a generic exception handler that reacts to a certain error condition:

```
#include <signal.h>
main ()
{
void handler (int sig);
...
signal (SIGINT, handler); /* SIGINT handler */
... /* do some processing */
if (error) raise (SIGINT); /* anomaly detected */
... /* continue processing */
}
void handler (int sig)
{
```

```
... /* handle error here */
}
```

In the C language, the signal library - function call is used to construct interrupt handlers to react to a signal from external hardware and to handle certain traps, such as fl oating - point overfl ow, by replacing the standard C library handlers.

PROCEDURAL LANGUAGES

Of the procedural languages discussed in this chapter, Ada has the most explicit exception handling facility. Consider an Ada exception handler to determine whether a square matrix is singular (i.e., its determinant is zero). Assume that a matrix type has been defi ned, and it can be determined that the matrix is singular. An associated code fragment might be:

```
begin
-- calculate determinant
-- ...
--
exception
when SINGULAR : NUMERIC/ERROR => PUT ("SINGULAR");
when others => PUT ("FATAL Error");
raise ERROR;
end;
```

Here, the exception keyword is used to indicate that this is an exception handler and the raise keyword plays a role similar to that of raise in the C exception handler just presented. The definition of SINGULAR , which represents a matrix whose determinant is zero, is defi ned elsewhere, such as in a header file.

Cardelli ' s Metrics and Procedural Languages

Taking the common set of procedural languages as a whole, Cardelli considered them for use in real - time systems with respect to his criteria. His comments are paraphrased in the foregoing discussion.

- Use variable in procedural language makes the compiler more efficient.
- Further, because modules can be compiled independently, compilation of large systems is efficient, at least when interfaces are stable
- In small - scale development makes testing ease and less debugging efforts.
- Finally, experienced programmers usually adopt a coding style that causes some logical errors to show up as type checking errors; hence, they can use the type checker as a development tool. For instance, changing the name of a type when its invariants change even though the type structure remains the same yields error reports on all its previous uses.

- Dependencies between such pieces of code are minimized, and code can be locally rearranged without any fear of global effects.
- Finally, procedural languages are economical because certain well – designed constructions can be naturally composed in orthogonal ways.

OBJECT - ORIENTED LANGUAGES

- The benefits of object - oriented languages, such as improved programmer productivity, increased software reliability, and higher potential for code reuse, are well known and appreciated.
- Object - oriented languages include Ada, C ++ , C#, and Java. Formally, object - oriented programming languages are those that support data *abstraction* , *inheritance* , *polymorphism* , and *messaging* .
- Objects are an effective way to manage the increasing complexity of real - time systems, as they provide a natural environment for information hiding, or protected variation and encapsulation. In encapsulation, a class of objects and methods associated with them are enclosed or encapsulated in class definitions.
- An object can utilize another object ' s encapsulated data only by sending a message to that object with the name of the method to apply. For example, consider the problem of sorting objects.
- Object - oriented languages provide a fruitful environment for information hiding; for instance, in image - processing systems, it might be useful to define a class of type pixel, with attributes describing its position, color, and brightness, and operations that can be applied to a pixel, such as add, activate, and deactivate.

Synchronizing Objects and Garbage Collection

Rather than extending classes through inheritance, in practice, it is often preferable to use composition.

Specifically, consider the following common synchronization policies for objects:

- *Synchronized Objects* . A synchronization object, such as a mutex, is associated with an object that can be concurrently accessed by multiple threads. If internal locking is used, then on method entry, each public method acquires a lock on the associated synchronization object and releases the lock on method exit. If external locking is used, then clients are responsible for acquiring a lock on the associated synchronization object before accessing the object and subsequently releasing the lock when finished.
- *Encapsulated Objects* . When an object is encapsulated within another object (i.e., the encapsulated object is not accessible outside of the

enclosing object), it is redundant to acquire a lock on the encapsulated object, since the lock of the enclosing object also protects the encapsulated object. Operations on encapsulated objects therefore require no synchronization.

- *Thread - Local Objects* . Objects that are only accessed by a single thread require no synchronization.
- *Objects Migrating between Threads* . In this policy, ownership of a migrating object is transferred between threads. When a thread transfers ownership of a migrating object, it can no longer access it. When a thread receives ownership of a migrating object, it is guaranteed to have exclusive access to it (i.e., the migrating object is local to the thread). Hence, migrating objects require no synchronization. However, the transfer of ownership does require synchronization.
- *Immutable Objects* . An immutable object ' s state can never be modified after it is instantiated. Thus, immutable objects require no synchronization when accessed by multiple threads since all accesses are read - only.
- *Unsynchronized Objects* . Objects within a single - threaded program require no synchronization.

Why parameterization of synchronization policies are necessary in Real-Time Systems?

Consider a class library that ensure the widest possible audience. for this library, so he makes all classes synchronized so that they can be used safely in both single - threaded and multi - threaded applications. However, clients of the library whose applications are single - threaded are unduly penalized with the unnecessary execution overhead of synchronization that they do not need. Even multi - threaded applications can be unduly penalized if the objects do not require synchronization (e.g., the objects are thread - local). Therefore, to promote reusability of a class library without sacrificing performance, classes in a library ideally would allow clients to select on a per - object basis which synchronization policy to use.

Garbage refers to allocated memory that is no longer being used but is not otherwise available either. Excessive garbage accumulation can be detrimental, and therefore garbage must be regularly reclaimed. Garbage collection algorithms generally have unpredictable performance, although average performance may be known. The loss of determinism results from the unknown amount of garbage, the tagging time of the nondeterministic data structures, and the fact that many incremental garbage collectors require that every memory allocation or deallocation from the heap be willing to service a page - fault trap handler.

Furthermore, garbage can be created in both procedural and object – oriented languages. For example, in C, garbage is created by allocating memory, but not deallocating it properly. Nonetheless, garbage is generally associated with object - oriented languages like C ++ and Java. Java is noteworthy in that the standard environment incorporates garbage collection, whereas C ++ does not.

Metrics and Object - Oriented Languages
Consider object - oriented languages in the context of Cardelli ' s metrics as paraphrased from his analysis.

- In terms of economy of execution, object -oriented style is intrinsically less efficient than procedural style. In pure object -oriented style, every routine is supposed to be a method. This introduces additional indirections through method tables and prevents straightforward code optimizations, such as inlining. The traditional solution to this problem (analyzing and compiling whole programs) violates modularity and is not applicable to libraries.
- With respect to economy of compilation, often there is no distinction between the code and the interface of a class. Some object - oriented languages are not sufficiently modular and require recompilation of super classes when compiling subclasses. Hence, the time spent in compilation may grow disproportionately with the size of the system.
- On the other hand, object - oriented languages are superior with respect to economy of small - scale development. For example, individual programmers can take advantage of class libraries and frameworks, drastically reducing their workload. When the project scope grows, however, programmers must be able to understand the details of those class libraries, and this task turns out to be more difficult than understanding typical module libraries. The type systems of most object - oriented languages are not expressive enough; programmers must often resort to dynamic checking or to unsafe features, damaging the robustness of their programs.
- In terms of economy of large - scale development, many developers are frequently involved in developing new class libraries and tailoring existing ones. Although reuse is a benefit of object - oriented languages, it is also the case that these languages have extremely poor modularity properties with respect to class extension and modification via inheritance. For instance, it is easy to override a method that should not be overridden, or to reimplement a class in a way that causes problems in subclasses. Other large - scale development problems include the confusion between classes and object types, which limits the construction of abstractions, and the fact that subtype polymorphism is not good enough for expressing container classes.
- Object - oriented languages have low economy of language features. For instance, C ++ is based on a fairly simple model, but is overwhelming in the complexity of its many features. Unfortunately, what started as

economical and uniform language (“ everything is an object ”) ended up as a vast collection of class varieties. Java, on the other hand, represents a step toward reducing complexity, but is actually more complex than most people realize.

Object - Oriented versus Procedural Languages

- There is no such rules/differences that whether Oops Language or procedural language boundary to be used in Real-Time Systems.

This is partially due to the fact that there is a huge variety of real - time applications — from nonembedded airline booking and reservation systems to embedded wireless sensors in running shoes, for example.

- The benefits of an object - oriented approach to problem solving and the use of object - oriented languages are clear.
- Moreover, it is possible to imagine certain aspects of a real - time operating system that would benefit from objectification, such as process, thread, file, or device. Furthermore, certain application domains can clearly benefit from an object - oriented approach. The main arguments against object - oriented programming languages for real - time systems, however, are that they can lead to unpredictable and ineffi cient systems, and that they are hard to optimize.
- Nonetheless, we can confidently recommend object - oriented languages for soft and fi rm real - time systems.
- The unpredictability argument is hard to defend, however, at least with respect to object - oriented languages, such as C ++ , that *do not use garbage collection*. It is likely the case that a predictable system — also a hard real – time one — can be just as easily built in C ++ as C. Similarly, it is probably just as easy to build an unpredictable system in C as in C ++ . The case for more unpredictable systems using object - oriented languages is easier to sustain when arguing about garbage - collecting languages like Java.
- In any case, the inefficiency argument against object - oriented languages is a powerful one. Generally, there is an execution - time penalty in object -oriented languages in comparison with procedural languages. This penalty is due in part to late binding (resolution of memory locations at runtime rather than at compile time) necessitated by function polymorphism, inheritance, and composition. These effects present considerable and often uncertain delay factors. Another problem results from the overhead of the garbage collection routines. One way to reduce these penalties is not to defi ne too many classes and only defi ne classes that contain coarse detail and high – level functionality.

Object - Oriented Languages Lack Certain Flexibility

The following anecdote (reported by one of Laplante ’ s clients who prefers

to remain anonymous) illustrates that the use of object - oriented language
for real - time systems may present subtle diffi culties as well. A design team
for a particular real - time system insisted that C ++ be used to implement a
fairly simple and straightforward requirements specifi cation. After coding
was complete, testing began. Although the developed system never failed,
several users wished to add a few requirements; however, adding these
features caused the real - time system to miss important deadlines. The client
then engaged an outside vendor to implement the revised design using a
procedural language. The vendor met the new requirements by writing the
code in C and then hand - optimizing certain assembly - language sections
from the compiler output. They could use this optimization approach
because of the close correspondence between the procedural C code and
compiler - generated assembly - language instructions. This straightforward
option was not available to developers using C ++ .

The vignette is not an endorsement of this solution strategy, however. It is
simply an illustration of a very special case. Sometimes such cases are used to
dispute the viability of object - oriented languages for real - time applications,
which is not fair — many punctual and robust real - time systems are built in
object - oriented languages. Moreover, while the client ' s problem was solved in
the straightforward manner described in the vignette, it is easy to see that the
understandability, maintainability, and portability of the system will be problematic.

Hence, the solution on the client ' s case should have involved a complete
reengineering of the system to include reevaluation of the deadlines and
the overall system architecture.

A more general problem is the inheritance anomaly in object - oriented
languages. The inheritance anomaly arises when an attempt is made to use
inheritance as a code reuse mechanism, which does not preserve substitutability
(i.e., the subclass is not a subtype). If the substitutability were preserved,
then the anomaly would not occur. Since the use of inheritance for reuse has
fallen out of favor in object - oriented approaches (in favor of composition), it
seems that most inheritance anomaly rejections of object - oriented languages
for real - time systems refl ect an antiquated view of object orientation.

Consider the following example from an excellent text on real - time operating
systems (Shaw, 2001):

```
BoundedBuffer
{
DEPOSIT
pre: not full
REMOVE
pre: not empty
}
```

```
MyBoundedBuffer extends BoundedBuffer
{
DEPOSIT
pre: not full
REMOVE
pre: not empty AND lastInvocationIsDeposit
}
```

Assuming that preconditions are checked and have “ wait semantics ” (i.e., wait for the precondition to become true), then clearly MyBoundedBuffer has strengthened the precondition of BoundedBuffer , and hence violated substitutability — and as such is a questionable use of inheritance.
Most opponents of object - oriented languages for real - time programming assert that concurrency and synchronization are poorly supported. However, when built - in language support for concurrency does not exist, it is a standard practice to create “ wrapper - facade ” classes to encapsulate system - concurrency
application program interface (API) for use in object orientation (e.g., wrapper classes in C ++ for POSIX threads). Furthermore, there are several concurrency patterns available for object - oriented real - time systems (Douglass, 2003 ; Schmidt et al., 2000). While concurrency may be poorly supported at the language
level, it is not an issue since developers use libraries instead.
In summary, critics of current object - oriented languages for real - time systems seem fi xated on Java, ignoring C ++ . C ++ is more suitable for real - time
programming since, among other things, it does not have built - in garbage collection
and class methods, and by default does not use “ dynamic binding. ” In any case, there are no strict guidelines when object - oriented approaches and languages should be preferred. Each specifi c situation needs to be considered individually.

OVERVIEW OF PROGRAMMING LANGUAGES

For purposes of illustrating the aforementioned language properties, it is useful to review some of the languages that are currently used in programming real - time systems. Selected procedural and object - oriented languages are discussed in alphabetical order, and not in any rank of endorsement or salient properties.

4.5.1 Ada

- Ada was first introduced DoD, USA and in 1983 it was standardized. Ada was intended to be used specifically for programming real - time systems, but, it is found the resulting executable code to be bulky and inefficient.
- Major problems exists when implemented for multitasking which fails. Ada 83 is renovated to newer version Ada 95 is considered the first

internationally standardized *object - oriented* programming language. Three particularly useful constructs were introduced in Ada 95 to resolve shortcomings of Ada 83 in scheduling, resource contention, and synchronization:

1. A pragma that controls how tasks are dispatched.
2. A pragma that controls the interaction between task scheduling.
3. A pragma that controls the queuing policy of task - and resource – entry queues.

Moreover, other additions to the language strived to make Ada 95 fully object - oriented. These included:

- Tagged types
- Packages
- Protected units

Proper use of these constructs allows for the construction of objects that exhibit the four characteristics of object - oriented languages.

In October 2001, a Technical Corrigendum to the Ada 95 Standard was announced by ISO/IEC, and a major Amendment to the international standard was published in March 2007. This latest version of Ada is called " Ada 2005. " The differences between Ada 95 and Ada 2005 are not extensive — in any case, includes a few changes that are of particular interest to the real - time systems community, such as:

- The real - time systems Annex contains additional dispatching policies, support for timing events, and support for control of CPU - time utilization.
- The object - oriented model was improved to provide multiple inheritance.
- The overall reliability of the language was enhanced by numerous improvements.

C

- The C programming language, invented around 1972 at Bell Laboratories, is a good language for " low - level " programming.
- The C language provides special variable types, such as register , volatile, static , and constant , which allow for effective control of code generation at the procedural language level.
- When variable used as a register type, the compiler to place such a declared variable in a work register, which often results in faster and smaller programs.
- Furthermore, C supports call - by – value only, but call - by - reference can be implemented easily by passing a pointer to anything as a value.
- Variables declared as type volatile are not optimized by the compiler at all. This feature is necessary in handling memory – mapped I/O and other special instances where the code should not be optimized.

- Automatic coercion refers to the implicit casting of data types that sometimes occurs in C.
- The C language provides for exception handling through the use of signals, and two other mechanisms, setjmp and longjmp , are provided to allow a procedure to return quickly from a deep level of nesting — a particularly useful feature in procedures requiring an abort.

- Overall, the C language is particularly good for embedded programming, because it provides for structure and flexibility without complex language restrictions.

C ++

- C ++ is a hybrid object - oriented programming language that was originally implemented as a macro extension of C in the 1980s.
- Today, C ++ stands as an individual compiled language & exhibits all characteristics of an object - oriented language and promotes better software - engineering practice through encapsulation and more advanced abstraction mechanisms than C.
- C ++ compilers implement a preprocessing stage that basically performs an intelligent search - and - replace on identifiers that have been declared using the #define or #typedef directives.
- The problem with the preprocessor approach is that it provides a way for programmers to inadvertently add unnecessary complexity to a program also it has weak type checking and validation.
- Previously C ++ programmers used complex pointer arithmetic to create and maintain dynamic data structures, cconsequently lots of bugs appear. Today, however, standard libraries of dynamic data structures are available.
- There are three complex data types in C ++ : classes, structures, and unions. However, C ++ has no built - in support for text strings.

- Multiple inheritance is a helpful feature of C ++ that allows a class to be derived from multiple parent classes also complicated to implement from the compiler perspective.
- C ++ still allows for low - level control; for instance, it can use inline methods rather than a runtime call.
- Furthermore, C ++ does not provide automatic garbage collection, which means dynamic memory must be managed manually or garbage collection must be homegrown.

- Therefore, when converting a C program to C ++ , a complete redesign is required to fully capture all of the advantages of an object - oriented design while minimizing the runtime disadvantages.

C#

- C# (pronounced “ C sharp ”) is a C ++ - like language that, along with its operating environment, has similarities to Java and the Java virtual machine, respectively.
- C# is first compiled into an intermediate language, which is then used to generate a native image at runtime.
- C# is associated with Microsoft’s .NET framework for scaled - down operating systems is highly configurable, capable of scaling from small, and upwards (e.g., for real - time systems requiring user - interface support).
- The minimum kernel configuration provides basic networking support, thread management, dynamic link library support, and virtual memory management.
- C# supports “ unsafe code, ” allowing pointers to refer to specific memory locations. Objects referenced by pointers must be explicitly “ pinned, ” disallowing the garbage collector from altering their location in memory. This capability could increase schedulability, and it also allows for direct memory access (DMA) to write to specific memory locations;
- Moreover, C# provides many thread synchronization mechanisms, but none with this level of precision.
- C# supports an array of thread - synchronization constructs: lock, monitor, mutex, and interlock. A Lock is semantically identical to a critical section — a code segment guaranteeing entry into itself by only one thread at a time like mutex .
- Finally, interlock, a set of overloaded static methods, is used to increment and decrement numeric in a thread - safe manner in order to implement the priority - inheritance protocol. Timers that are similar in functionality to the widely used in C#. The accuracy of these timers is machine dependent, and thus not guaranteed, reducing their usefulness in real - time systems to be used in multiple hardware platforms.
- C# and the .NET platform are not appropriate for the majority of hard real -time systems for several reasons, including the unbounded execution of its garbage - collection environment and its lack of threading constructs to adequately support schedulability and determinism.
- C# ’ s ability to interact effectively with operating - system APIs, shield developers from complex memory management logic, together with C# ’

s good floating – point performance, make it a programming language that is highly potential for soft and even firm real - time applications.

Java

- Java, in the same way as C#, is an interpreted language, that is, the code compiles into machine - independent intermediate code that runs in a managed execution environment.
- The obvious advantage is that Java code can run on any device that implements the virtual machine.
- However, there are also native - code Java compilers, which allow Java to run directly “ on the bare metal, ” that is, the compilers convert Java to assembly code or object code.
- Furthermore, there are even special Java microprocessors, which directly execute Java byte code in hardware .
- Java is an object - oriented language and the code appears very similar to C++. Java supports call - by - value. But Java is a pure object - oriented language, that is, all functionality in Java has to be implemented by creating object classes, instantiating objects of those classes and manipulating objects ’ attributes through methods.
- Of course, a good object - oriented design is not guaranteed, but the design obtained in the conversion will be a true object - oriented one based on the rules of the language. This situation is quite different from the kind of false object - oriented conversion that can be obtained from C to C ++ in the blunt manner previously highlighted.
- Java does provide a preprocessor but constant data members are used in place of the #define directive, and class definitions are used in lieu of the #typedef directive.
- The Java language does not support pointers, but it provides similar functionality via references. Java passes all arrays and objects by reference, which prevents common errors due to pointer mismanagement.
- Java only implements one complex data type: classes. Java programmers use classes when the functionality of structures and unions is desired. This consistency comes at the cost of increased execution time over simple data structures.
- The Java language does not support standalone functions. Instead, Java requires programmers to bundle all routines into class methods again with significant cost. Moreover, Java has no direct support for multiple inheritance however, allow for implementation of multiple inheritance.
- In Java, strings are implemented as first - class objects meaning that they are at the core of the Java language. Java ’ s implementation of strings as objects provides several advantages. First, string creation and access is

consistent across all systems. Next, because the Java string classes are defi ned as part of the Java language strings function predictably every time. Finally, the Java string classes perform extensive runtime checking, which helps eliminate errors. But all these operations increase execution time.

- Operator overloading is not supported in Java. However, in Java's string class, " + " represents concatenation of strings, as well as numeric addition.
- The Java language does not support automatic coercions. In Java, if a coercion will result in a loss of data, then it is necessary to explicitly cast the data element to the new type.

Real - Time Java

- Real – time Java is just a modification of the standard Java language, because it is used increasingly in implementing soft, firm, and even hard real - time systems, while the standard Java is mainly used for soft real - time systems only.
- In addition to the unpredictable performance of garbage collection, the Java specification provides only broad guidance for scheduling.
- This preference is not, however, a guarantee that the highest - priority one of ready threads will always be running, and thread priorities cannot be used to reliably implement mutual exclusion. It was soon recognized that this and other shortcomings rendered standard Java inadequate for most real – time systems.

requirements for the real - time specification of Java (RTSJ 1.0) are :

R1. The specification must include a framework for the lookup and discovery of available profiles.

R2. Any garbage collection that is provided shall have a bounded preemption latency.

R3. The specification must defi ne the relationships among real - time Java threads at the same level of detail as is currently available in existing standards documents.

R4. The specification must include APIs to allow communication and synchronization
between Java and non - Java tasks.

R5. The specification must include handling of both internal and external asynchronous events.

R6. The specification must include some form of asynchronous thread termination.

R7 . The core must provide mechanisms for enforcing mutual exclusion without blocking.

R8 . The specification must provide a mechanism to allow code to query whether it is running under a real - time Java thread or a nonreal - time

Java thread.
R9 . The specification must defi ne the relationships that exist between real - time Java and nonreal - time Java threads.

- The RTSJ defines the real - time thread class to create threads which can access objects on the heap and d therefore can incur delays because of garbage collection.
- For garbage collection, the RTSJ extends the memory model to support memory management which ensures no interfere with the real - time code' stability to provide deterministic behaviour.
- RTSJ uses " priority " somewhat more loosely than is traditionally accepted. " Highest priority thread " merely indicates the most eligible thread — the thread that the scheduler would choose from among all threads ready to run.

If the active scheduling policy permits threads with the same priority, the threads are queued using the FIFO principle. Specifically, the system (1) orders waiting threads to enter synchronized blocks in a priority queue; (2) adds a blocked thread that becomes ready to run to the end of the ready queue for that priority; (3) adds a thread whose priority is explicitly set by itself or another thread to the end of the ready queue for the new priority; and (4) places a thread that performs a yield to the end of its priority queue. The priority - inheritance protocol is implemented by default. The real - time specifi cation also provides a mechanism by which a systemwide default policy can be implemented.

*[The asynchronous event facility comprises two classes: AsyncEvent (acts like a hardware interrupt and AsyncEventHandler is a schedulable object roughly similar to a thread. Unlike other runable objects, however, an AsyncEventHandler has associated scheduling, release, and memory parameters that control the actual execution of read or write.
Asynchronous control transfer allows for identification of particular
methods by declaring them to throw an AsynchronouslyInterrupted
Exception (AIE). When such a method is running at the top of a thread ' s
execution stack and the system calls java.lang.Thread.interrupt() on
the thread, the method will immediately act as if the system had thrown an
AIE. If the system calls an interrupt on a thread that is not executing such a
method, the system will set the AIE to a pending state for the thread and will
throw it the next time control passes to such a method, either by calling it or
returning to it. The system also sets the AIE ' s state to " pending " while control
is in, returns to, or enters synchronized blocks.
The RTSJ defi nes two classes for programmers who want to access physical
memory directly from Java code. The fi rst class, RawMemoryAccess , defi nes
methods that let you build an object representing a range of physical addresses
and then access the physical memory with byte , word , long , and multiple

byte granularity. The RTSJ implies no semantics other than the set and get methods. The second class, PhysicalMemory , allows the construction of a PhysicalMemoryArea object that represents a range of physical memory addresses where the system can locate Java objects. For example, a new Java object in a particular PhysicalMemory object can be built using either the newInstance() or newArray() methods. An instance of RawMemoryAccess models a raw storage area as a fi xed - size sequence of bytes. Factory methods allow for the creation of RawMemoryAccess objects from memory at a particular address range or using a particular memory type. The implementation must provide and set a factory method that interprets these requests accordingly. A full complement of get and set methods lets the system access the physical memory area ’ s contents through offsets from the base — interpreted as byte , short , int , or long data values — and copy them to or from byte , short , int , or long arrays.]

COMPILER OPTIMIZATIONS OF CODE

- There exist infinitely many object codes that implement the same computations produce the same outputs when presented with the same inputs.
- Some of these object codes may be faster while others may require less memory. Hence, it is more important use an optimized code.
- When beginning to use a new compiler, it is important to expose the details of the compiler for effective optimized code design as well as one should know both the language and your compiler thoroughly.
- Moreover, many of the techniques used in code optimization underscore, but the fact that mathematical technique is that reformulate any algorithm or expression to eliminate time - consuming function calls to improve real - time performance.
- Most of the code optimization techniques used by compilers can be exploited to reduce response times. Often these strategies are employed invisibly by the compiler, or can be turned on or off with compiler directives or switches.
- Optimization techniques that uses different techniques are
  - Use of arithmetic identities and intrinsic functions
  - Reduction in strength
  - Common subexpression elimination
  - Constant folding

- Loop invariant removal & induction elimination
- Use of registers and caches
- Dead - code removal
- Flow - of - control optimization
- Constant propagation
- Dead store elimination & variable elimination
- Short - circuit Boolean code
- Loop unrolling & jamming
- Cross - branch elimination

Standard Optimization Techniques

- Good compilers *use arithmetic identities* to eliminate useless code.
- *Reduction in strength* refers to the use of the fastest machine – language instructions possible to accomplish a given operation. For example , when optimizing for speed, replace multiplication by a series of shift operations. Shift instructions are faster than integer multiplication in certain CPU environments.
- Care should therefore be taken in deciding whether a particular variable should be defined as a character or an integer.
- It is well known that division instructions typically take longer to execute than multiplication instructions, it is better to multiply by the reciprocal of a number than to divide by that number.
- *Repeated calculations of the same subexpression* in two different expressions should be avoided.
- *use intrinsic functions* rather than ordinary functions as Intrinsic functions are simply macros where the actual function call is replaced by inline code during compilation. This improves real - time performance/
- Most compilers perform *constant folding* , but this should not be assumed when beginning to use a new compiler. (i.e. precomputing some values before compilation)
- Most compilers will move such computations outside loops that do not need to be performed within the loop, a process called *loop invariant removal* .

- This moves an addition outside the loop, but again requires more memory.
- An integer variable i is called an *induction variable of a loop* if it is incremented or decremented by some constant on each cycle of the loop. A common situation is one in which the induction variable is i and another variable, j , which is a linear function of i , is used to offset into some array. Often i is used solely for a test of loop termination. In such case, variable i can be eliminated by replacing its test for one on j instead. For example, consider the following C program fragment:

  ```
  for (i =1;i<=10;i++)
          a[i+1]=1;
  ```

  An optimized version is:

  ```
          for (j =2;j<=11;j++)
          a[j]=1;
  ```

  eliminating the extra addition within the loop.
- When programming in assembly language or when using languages that support register - type variables, calculations using *work registers is preferable*. Typically, register - to - register operations are faster than register - to - memory ones.
- If the CPU architecture supports *memory caching* , for frequent use of variables cache based variables is better to use.
- One of the easiest methods for decreasing memory utilization is to *remove dead or unreachable code* — that is, code that can never be reached in the normal flow - of - control. Such code might be debug instructions that are executed only if a debug flag is set, or some redundant initialization instructions.
- In a microcontroller environment, debugging takes a lots of time so debug code should preferably be implemented using the conditional compile facilities available with most compilers.
- Certain variable - assignment expressions can be changed to *constant assignments*, thereby permitting rasterization opportunities or the use of faster immediate addressing mode.

- Variables that contain the *same value within a short segment of code* can be combined into a single temporary variable. For example,
- A variable is said to be alive at a point in a program if its value can be used subsequently; otherwise it is *dead and subject to removal* .

- If the same code appears in more than one case in a case or switch statement, then it is better to combine such cases into one. This *eliminates an additional branch or cross branch* .

Additional Optimization Considerations

• *Arrange entries in a table* so that the most frequently sought values are the fi rst to be compared.

• *Replace threshold tests on monotone functions* (continuously decreasing or increasing) *by tests on their parameters* , thereby avoiding evaluation of the function itself. For instance, if exp(x) is a function computing *ex* , then instead of using:

```
if (exp(x) < exp(y)) then ...
```

use:

```
if (x < y) then ...
```

which will save two evaluations of the costly function exp() .

• *Link the most frequently used procedures together* to maximize the locality of reference (applies only in cached or paging systems).

• Store procedures in memory in sequence so that calling and called procedures will be loaded together to increase the locality of reference.

Again, this only applies in cached or paging systems.

• *Store redundant data elements close to each other* to increase the locality of reference (applies only in cached or paging systems).

Even though many of the optimization techniques discussed above can be and have been automated, some compilers only perform one optimization pass, overlooking opportunities that are not revealed until after at least a single pass. Hence, manual optimization may provide additional execution - time savings. To see the *cumulative effects of multiple - pass optimization* , consider the following example.

Example: Multiple - Pass Optimization

Begin with the nonoptimized C - code fragment:

```
for (j =1;j<=3;j++)
{
a[j]=0;
a[j]=a[j]+2*x;
}
for (k =1;k<=3;k++)
b[k]=b[k]+a[k]+2*k*k;
```