ABSTRACT

Transformations and Analysis of Parallel Real Time Programs

by

Chandima J. Gunasekara

The problem of schedulability analysis of a set of real time programs form a NP complete problem. The exponential complexity of analysis is a direct result of the complexity in the real time programs, as a combinatorial explosion takes place when trying to determine access patterns of shared resources. Thus, to transform the original programs to a less complex form, while preserving its timing characteristics, is the only viable solution. By using such transformations to reduce the complexity of real time programs, it is possible to schedulability analyze programs at compile time efficiently, without adding an unnecessary overhead to the compilation time. A set of suitable transformations and run time scheduling algorithms are introduced and implemented in C++. A library of transformations and analysis routines are provided. The library routines can be used to build prototype schedulability analyzers for testing various analysis techniques. These transformations and the scheduling algorithm will be an integral part of the real time compiler for the real time language RTL. The RTL compiler will not only produce fast and efficient code for an arbitrarily specified real time hardware architecture, but also will provide the worst case timing characteristics for the programs.

Transformations and Analysis of Parallel Real Time Programs

by
Chandima J. Gunasekara

APPROVAL PAGE

Transformations and Analysis of Parallel Real Time Programs

by

Chandima J. Gunasekara

Dr. Alexander D. Stoyenko
Assistant Professor
Department of Computer and Information Science, NJIT

BIOGRAPHICAL SKETCH

Author: Chandima J. Gunasekara

Degree: Master of Science in Computer Science

Date: May, 1992

Date of Birth:

Undergraduate and Graduate Education:

- Master of Science in Computer Science, New Jersey Institute of Technology, Newark, NJ, 1992

- Bachelor of Science in Computer Science, University of Wales College of Cardiff, Cardiff, United Kingdom, 1990

Major: Computer Science

ACKNOWLEDGEMENT

TABLE OF CONTENTS

1. Introduction

The use of embedded real time computing systems for control are rapidly growing. A failure of an embedded computing systems to properly control its real time processes may lead to major economic loses, (including human life). Thus, a real time application demands from its computing system, a guarantee of predictable, reliable and timely operations. Real time computing systems with predictable behavior can indeed be realized. Requirements of predictable system behavior, given time-constrained, functional specification of the environment, can be embodied into the, programming language, operating system and hardware. The resulting real time computing system is subjectable to an a priori assessment of predictable behavior; which is referred to as schedulability analysis [5].

The construction of a embedded hard real time computing system requires a priori knowledge of deadline satisfaction of the tasks to be scheduled. Thus the construction of real time software and its implementation must be done on a deterministic hardware and software platform. Most often it is the case that real time software is designed on hardware and software environments that are oblivious to the higher level software designer, causing the estimation of the execution times to be over pessimistic upper bounds or average statistics. Most often extensions to existing hardware architectures are proposed as support processors that implement computationally expensive features of the programming language in order to achieve higher execution speeds [3]. The main issue of real time computation is not increasing execution speed or minimizing response time, but *deadline satisfaction* [4].

Real time operations distinguish itself from other forms of data processing by the explicit involvement of the dimension of time. Real time systems must fulfill the

following user requirements under all, including extreme, load conditions.

- timeliness,

- simultaneity,

- predictability, and

- dependability.

Timeliness can be viewed as deadline satisfaction, i.e. given a set of dead lines, it is the task of the software designer to design a set of software modules (processes) that conform to the given timing requirements. Thus, the software designer should have the freedom to specify and configure a hardware architecture that suit the application, and not have to alter the problem specification to suit the hardware architecture. Real time systems must be distributed and must provide parallel processing capabilities, giving rise the need for simultaneous processing. Predictability is central in order to achieve timelyness, thus requiring every component in the real time computing system to be deterministic. The dependability requirement deals with the issues of fault tolerance both at the hardware and software levels. A more detailed discussion on the above mentioned requirements can be found in [5].

The real world is inherently non deterministic, i.e. there is no uniform methodology to model all phenomena, relationships and events that occur in the environment. Certainly, there is no general formula that can be applied to predict future outcomes in a deterministic manner. Thus, all real time systems must be a subset of a non real time system. Moreover, a real time system will only be allowed to interact with a *restricted* environment, making it possible for deterministic behavior.

Fig. 1.1 shows a diagram of the conceptual model of a typical real time system. This system is comprised of a real time computing subsystem that interacts with the real time external environment and a non real time computing subsystem that interacts with the non real time external environment. These two subsystems are connected via a communication port. The real time computing system may consist of an arbitrary number of processes connected by a communication network, that allows communication between the processors and interfaces to the environment that control or receive information. The non real time computing system is a general purpose off the shelf computer (e.g. Vax or Sun system) with a VMS or UNIX operating system. The communication port is used for non time critical message passing between the two subsystems. The non real time computing system will act as a host that will initiate a real time task and monitor its progress until termination of that task.
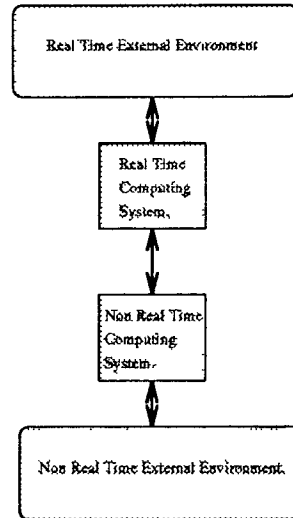
Fig 1.1. Conceptual model of a real time system.

A real time programming language must be schedulability analyzable [4]. Schedulability analysis is the task of (manually or automatically) finding execution times for, statements, processes, and modules that are to be used in a system that requires real time guarantees from the software. Thus one of the main criteria for a real time system is to have a very deterministic hardware architecture. Also, a real time programming language should have no *dynamic data structures, recursive function calls,* both direct or indirect. All loop constructs should have compile time known bounds. The above mentioned requirements are the most important and only apply to real time programming languages.

Constructing a schedulability analyzable programming language and a predictable hardware architecture will be our focus of attention. Work has been done on this area in [1],[2],[4] [5] and [6]. The most significant contribution is in [5], where a real time programming language (RT Euclid) and its associated hardware architecture is proposed. A subset of RTE is implemented along with its schedulability analyzer.

Schedulability analysis on a language is a NP-complete problem. Thus, compilation of a program or a set of programs will incur an overhead due to the schedulability analysis stage. The overhead on the compiling process is a direct result of the complexity of the real time programs being compiled. In [1] a methodology is proposed for a polynomial-time transformation that can be performed on the programming language in order to improve the performance of the schedulability analysis stage. However, the transformations in [1] are only defined for a very restricted subset of RTE that only allows the manipulation of a single resource. *In this research project the objective is to extend the language to a more general form allowing for multiple resources.* Consequently, the transformations introduced in [1] and the analysis algorithm are extended appropriately to take account of the extended language.

1.1 Related Previous Work

An extensive amount of work has been done in the past in real time languages and real time scheduling. Most of the work are based on over simplistic assumptions and much more research work is needed in this area in order to provide a comprehensive system for hard real time software development. The research effort is broadly focused on, 1) specification and verification, 2) real time scheduling theory, 3) real time operating systems, 4) real time programming languages and design methodology, and 5) fault tolerance. While there are many contributions in each of the above mentioned areas what is needed is to unify all the research results in the above mentioned areas in order to provide a comprehensive integrated system for real time software development.

In [2] a methodology for specifying and proving assertions about time in higher-level language programs is described. The methodology introduces three ideas. 1) The distinction between, and the treatment of, both computer and real time in the system. 2) The use of upper and lower bounds on the execution time of the program elements. 3) A simple extension of Hoare logic to include the effects of the passage of real time. Thus, it is based on finding the best and the worst execution time bounds for statements. Since the timing analysis is mostly based on the language, independent of the underlying hardware and the runtime support system, the applicability of this methodology is limited.

In [8] it is proved that finding a feasible non preemptive schedule for the following two problem classes is NP complete.

1. at least two CPUs, no resources, constant maximum response times, infinite frames, constant CPU requirements, no IPC and o overhead.

2. at least two CPUs, no resources, constant maximum response times, infinite frames, CPU requirements of 1 or 2 time units only, arbitrary IPC and no overhead.

Thus, finding heuristics for scheduling and schedulability analysis is more feasible than trying to compute the exact solutions in the most efficient manner. However, when heuristics are used the solution becomes a approximation to the exact case and needs some form of justification of the error.

In [9] a model with a, a single CPU, no resources, constant maximum response times, constant CPU requirements, no IPC, and no overhead, is analyzed. This model and the results obtained in [9] provide a good foundation for future work. However is too simple for most practical real time systems.

In [6] a model with, a single CPU, resources and resource contention, constant maximum response times, frames equal to their corresponding maximum response times, and processes consisting of sequential segments is analyzed. Also, it is assumed that, each segment has its own constant CPU and resource requirements, some resources require mutual exclusion, there is no IPC, processes are scheduled by the preemptive earliest deadline first, policy. Resources are allocated FCFS, and a segment cannot proceed until all its resources are allocated. In this model each task has a fixed set of requirements which may be needed by its segments. The analysis

provides a worst case analysis of the tasks. However, the algorithm is of polynomial complexity and the results are overly pessimistic guaranteed response times. The schedulability analysis results in a set of very coarse worst case response time bounds and can be improved.

In [4] a real time language, Real Time Euclid (RTE), is introduced. This language is specifically designed to address reliability and guaranteed schedulability issues in real time systems. Real Time Euclid employs exception handlers and import/export lists to provide comprehensive error detection, isolation, and recovery. The RTE language definition forces every construct in the language to be time- and space- bounded, i.e. the language has no dynamic data structures, recursive function calls, and all loop constructs have compile time known loop bounds. In later work by the authors in [4] a compiler/schedulability analizer is implemented for RTE. However, due to the NP complete nature of schedulability analysis the compiler/schedulability analyzer incurs a large amount of time in order to provide the schedulability results. Thus, more work is needed in order to make the schedulability analysis more interactive with the users.

In [1] a static semantic analysis and transformations on a restricted subset of RTE is introduced, and is shown how it can be used in a limited language to produce simple analyzable program forms. Also in [1] static analysis is combined with program transformations, reducing the cost of analysis. Also, a restricted form of shared resource contention of processes is introduced. The basic notion is that (1) all resource requests participating in a non idling resource interval are released together, when the last request is finished, and (2) the resource scheduler enforces statically pre computed non idling resource interval sizes.

In this thesis the work presented in [1] is extended to include more general

language constructs, such as nested conditionals and the manipulation of multiple resources. The static semantic analysis and the transformations are integrated in a prototype. Also, a library of routines in C++ for transformations and schedulability analysis are given. This library can be used as a testbed for testing new transformations and various heuristics for analysis.

2. Transformations and Schedulability Analysis

Plynomial time transformations of real time programs used in conjunction with schedulability analysis, significantly increase the class of real time programs that may be analyzed efficiently for guaranteed schedulability at compile time [1]. The accuracy of the results obtained form the schedulability analyzer will depend on the *type* of transformations used. Obviously, tools or analytic techniques which would allow efficient analysis of larger class of programs, or reduce the expected time of inefficient techniques, would be highly desirable. At one end of the spectrum are the schedulability analysis techniques that are highly precise yet not practicable due to the combinatorial execution orders. At the other end of the spectrum are the schedulability analysis techniques that are highly imprecise with polynomial execution orders. In between are the techniques that are some what precise with reasonable execution orders. The techniques that are proposed in our work are in the latter category of schedulability analysis techniques.

Moreover, important questions, relating to the above mentioned type of schedulability analysis techniques are, how precise are these techniques in practise? and what are their execution orders? How much of an overhead is imposed on the compilation stage as a result of schedulability analysis? Most importantly, if transformations are used, what is the expected improvement in the schedulability analysis stage?, i.e. how efficient are the transformations in reducing the original problem size. The problem size can be characterized as the amount of "time" that needs to be expended by the schedulability analyzer in order to yield the desired results. In this thesis an attempt is made to answer some of the above mentioned questions.

There are four types of transformations, viz, *deadline isomorphic, deadline preserving, deadline extending* and *deadline distroying.* These four types are defined as:

p: Transformed programs meets deadlines.
q: Original program meets deadlines.

A. **Deadline isomorphic:** $p \Leftrightarrow q.$
B. **Deadline preserving:** $q \Rightarrow p.$
C. **Deadline extending:** $p \Rightarrow q.$
D. **Deadline destroying:** $\neg A \wedge \neg B \wedge \neg C.$

deadline preserving or deadline isomorphic transformations will not be capable of reducing the problem size of schedulability analysis for all programs, i.e. some programs may resist transformations of types A and B. However, in such a case, a deadline extending transformation my be used effectively to reduce the problem size. One must remember that simply reducing the problem size is not our only objective, but also to provide valid schedulability results. A deadline extending transformation may cause the transformed program to violate the specified deadline when the original program would have met the specified deadline. Thus, it is up to the user to decide when not to use deadline extending transformations.

2.1 The Language Model

The subset of the language that is considered is given in Fig. 2.1. In this language definition, only the language constructs that are of importance are given, i.e. the language constructs that are important for schedulability analysis. The language RTE-1 is a subset of RTE consisting of a static number of top level processes and procedures with sequences of statements, conditionals and loops. The conditionals may be nested arbitrarily[1]. All loops are for-loops with a compile time knowable

---

[1] In an actual implementation of the RTE-1 language it is feasible to allow only for a constant depth of nesting.

loop bounds. The monitors may have multiple entries. Multiple monitor entries are modeled as a collection of critical sections, with the property that if a process is executing inside any one section, another process requesting entry to this or another section is blocked until the executing process exits the section. A process may be periodic or aperiodic depending on the definition. If a process is periodic then it will be activated once per every frame time.

program:

> *process_name* **frame** *positive_integer* **periodic**$_{opt}$
> *variable_declarations*
> *statement*

statement:

> *variable_name* = *expression;*
> **section** *section_id(operation, parameter_list);*
> *compound_statement*
> *selection_statement*
> *iteration_statement*

compound_statement:

> { *statement_list* }

statement_list:

> *statement*
> *statement statement_list*

iteration_statement:

> **for** *range* **do** *statement*

selection_statement:

> **if** (*expression*) *statement*
> **if** (*expression*) *statement* **else** *statement*
> **switch** (*expression*) { *switch_statement* }

switch_statement:

> *switch_statement_list default_statement*$_{opt}$

switch_statement_list:

> *labeled_statement* **break**$_{opt}$
> *labeled_statement* **break**$_{opt}$ *switch_statement_list*

labeled_statement:

> **case** *constant_expression* : *statement*

default_statement:

> **default** : *statement*

Fig. 2.1. Programming language definition for RTE-1.

2.2 The Hardware Model

The ideal is to be able to specify an arbitrary hardware architecture to the schedulability analizer such that it is reflected in the schedulability results of a set of processes. However, we restrict our model to a system with n processors and a ring communication network topology connecting the processors. Each processor is capable of executing at most one RTE-1 process. All accesses to resources are done via monitors that communicate across the network. Also, whenever a resource is declared in the system a dedicated processor is used to handle the requests to that resource. Each resource access request is queued up at the processor servicing that resource. The requests to a particular resource may arrive in an arbitrary order i.e. without any causal ordering of the requests with respect to each other.

Also, in the above describe model each process is assumed to have the same priority. In future extensions to the system one may consider processes with pre-assigned priorities when arbitrating resource requests. The software routines that are provided for analysis can be easily extended to take account of process priorities.

3. Transformations of Real Time Programs with k-way Conditionals with d Levels of Nesting

The RTE-1 language allows the programmer to nest conditionals arbitrarily (see the language definition given in fig. 2.1), thus increasing the cost of schedulability analysis. The objective is to take a RTE-1 program and transform it as much as possible such that the cost of schedulability analysis will be minimal. In this section, the transformations of types A, B and C are defined. Also, an algorithm is presented for applying the transformations recursively, to a program flow graph (PFG) representing a RTE-1 program.

The front end of the schedulability analyzer builds a PFG, using a attribute grammar like process, that closely corresponds to the parse tree of the program being compiled. The PFG contains the information needed to produce the object code while the transformations are applied. A PFG is obtained after clustering the program in to blocks of simple segments and critical segments and unroling of loops. Once the transformed PFGs are obtained for a set of processes the schedulability analyzer produces a static schedule resolving the delays associated with critical sections. The executable files are created using the resolved delays obtained from the static schedules. This scenario is shown in fig. 3.1.

3.1 The Structure of a PFG

A PFG is a DAG consisting of critical segment nodes (C), Simple segment nodes (S), Fork nodes (F) and Join nodes (J). Each node (segment) consists of a type (T), ($T \in \{C, S, F, J\}$), execution time (t) for that segment, depth (d) at the point in the PFG where a particular node occurs, the number of branches (n_b) with a list

14
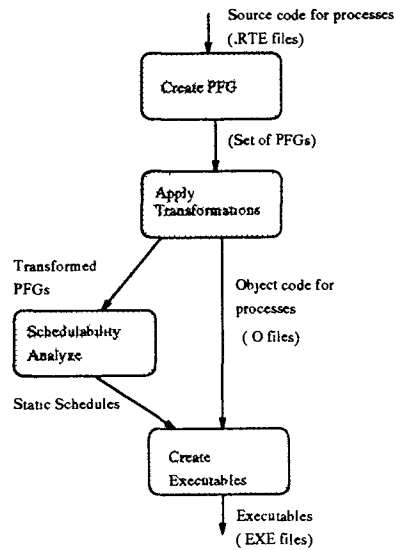
Fig. 3.1. The compilation/schedulability analysis process.

of pointers to the branches and a reference count r_c. In case of a C, S or J node, $n_b = 1$. In the case of a C, S or F node $r_c = 1$. In a C node the additional fields id, Δ and r_t are used to represent the resource id, a delay and the release time respectively. The purpose of Δ will be explained later. For the purpose of testing the transformations PFGs may be created using the grammar given in fig. 3.2. Thus it is possible to construct a PFG as a text file and apply transformations to it. The representation of a PFG given in fig. 3.2 is used as an intermediatory form of representation. This approach simplifies the task of testing the transformations and the analysis algorithms.

<PFG>	::=	<Tuples>
<Tuples>	::=	(<Tuple>) <Tuples> | (<Tuple>) |
		(F,0) { <Branches> } (J,0) | ϵ
<Branches>	::=	[<Tuples>] <Branches> | [<Tuples>]
<Tuple>	::=	(C,t,Resource_id) | (S,t)

Fig. 3.2. The grammar for generating a PFG.

3.2 Definition of the Transformations

Let $Sequence_{i,j}$ represent the jth segment on branch i, where $i = 0, 1, \cdots, n_b - 1$ and $j = 0, 1, \cdots, length_i - 1$ where $length_i = length\_C_i + length\_S_i$, where $length\_C_i$ is the number of critical segments on branch i and $length\_S_i$ is the number of simple segments on branch i. Also let $time_i = time\_C_i + time\_S_i$, where $time\_C_i$ is the sum of the execution time of critical segments in branch i and $time\_S_i$ is the sum of the execution time of simple segments in branch i.

Domination of branches:

Consider two alternate branches p and q. Branch p is said to *dominate* branch q if it is possible to pad q with idling delays such that it is time-wise equivalent to p. Thus, after padding q, taking branches p or q at execution time will have the same net time-wise effect. Therefore, if p dominates q then it will suffice to only look at branch p in order to compute the delays associated with accessing critical sections. Consequently, when p doesn't dominate q and q doesn't dominate p, they are considered to be *irreducible* branches.

In the transformations given below, if branch p dominates branch q then simply *eliminate* q from the PFG. However, when generating object code it is necessary to pad with the appropriate delays whenever a branch is eliminated.

The function Apply_Transformation() in Alg.1 choses a suitable transformation (given below) and applies it to a pair of branches eliminating a branch if possible. This continues until no further branches can be eliminated.

Transformation 1:

if $length_q = 0$ then eliminate q.

Transformation 2:

if $length\_C_p = length\_C_q = 0$ and $time_p \geq time_q$ then eliminate q.

Transformation 3:

if $length\_C_q = 0$ and $time_p \geq time_q$ then eliminate q.

Transformation 4:

if $length_p \geq length_q$ and $time_p \geq time_q$ and $Sequence_{p,j}.T = Sequence_{q,j}.T$ and $Sequence_{p,j}.t \geq Sequence_{q,j}.t$ and $Sequence_{p,j}.id = Sequence_{q,j}.id$ for $j = 0, 1, \cdots, length_p - 1$ then eliminate q.

The transformations given above are deadline isomorphic (type A) and deadline preserving (type B). Also, it is possible to construct transformations of types A and B that move code from simple segments to critical segments such that one branch will dominate. Note that the opposite is not possible. For example consider the PFGs given in fig. 3.3. The PFG in fig. 3.3 (a) can be transformed in to the PFG in fig. 3.3 (b) by moving one unit of code from s_3 to c_2 and one unit of code from s_4 to c_2. The resulting PFG given in fig. 3.3 (b) can now be transformed using a deadline preserving transformation given above.

Consider the PFGs given in fig.3.4 (a). The simple segment code s_1 can be split in to a simple segment and a critical segment as shown in fig. 3.4 (b). By applying transformation 4 to the PFG in fig. 3.4 (b) it is possible to reduce the right branch. This transformation is deadline extending and may cause the program to miss its specified deadline. Whether a program misses its deadline or not, as a result of a type C transformation, will depend on how much slack exists in the original program. Also, consider the PFG in fig. 3.5 (a), which is clearly a irreducible conditional.
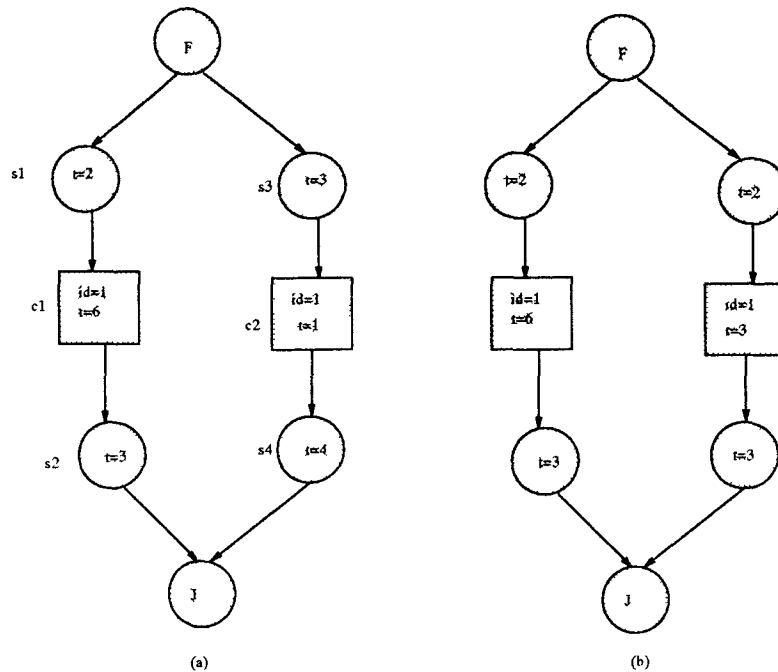
Fig. 3.3. An example of a deadline preserving transformation by moving code form a simple segment to a critical segment.

However, by moving three units of code from s_1 and one unit of code from s_2 in to the critical segment c_1, it is possible to transform it to the PFG given fig. 3.5 (b). Then, by applying transformation 4 it is possible to reduce the right branch of the PFG in fig. 3.5 (b). Since this transformation required both its branches to be adjusted it is deadline extending.
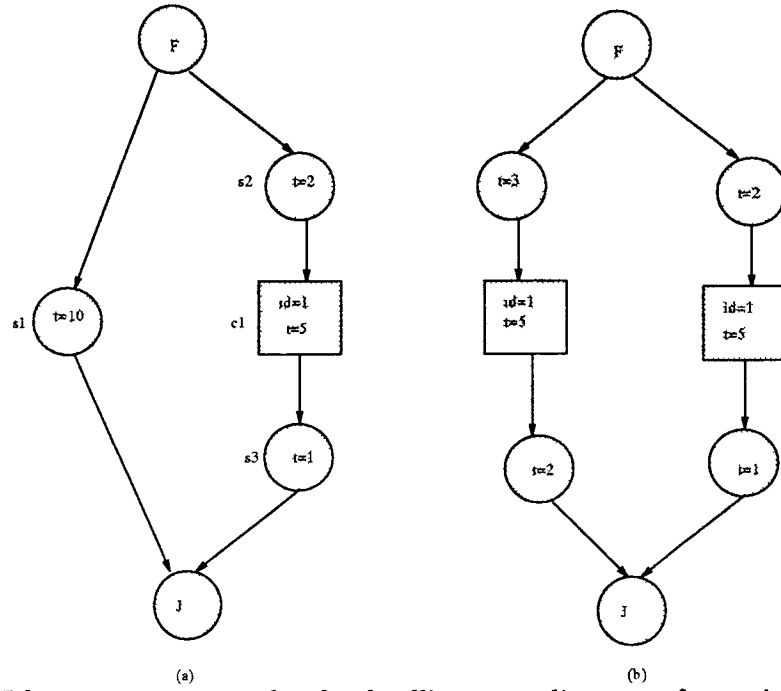
Fig. 3.4. An example of a deadline extending transformation.
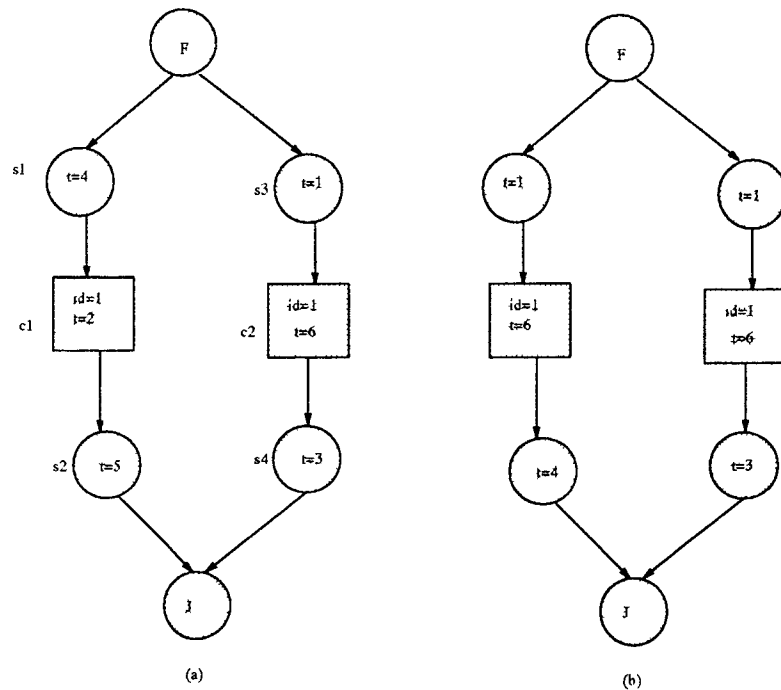
Fig. 3.5. An example of a deadline extending transformation on a irreducible conditional.

3.3 Algorithms for the Transformations

The algorithms given in Alg. 3.1 transforms a PFG to a less complex form by start-
ing at maximum depth and applying the transformations recursively to each k-way
conditional in the PFG. Also, it is assumed that before applying Alg. 3.1 to a PSG
it is preclustered. Preclustering a PFG means taking take a sequence of simple seg-
ment nodes and replacing it with a with a single simple segment node that has
the same total time. For example the following sequence of simple segment nodes
$(S, t_1), (S, t_2) \cdots, (S, t_p)$ can be replaced by $(S, \sum_{i=1}^{p} t_i)$.

```
boolean PFG::Transform(Node *n) {
boolean Tr,Trfm = TRUE;
    while (n ≠ Null)
        switch (n→type)
          case S:
          case C: n = n→get_next
             break
          case F:
              Tr=TRUE
              for ( each branch do )
                 Tr = Tr ∧ Transform(branch)
              if (Tr)
                 Tr = Apply_Transformation(n)
              else if ((n → n_b) > 2)
                 Tr = Apply_Minimize(n)
              if (Tr)
                 Temp = n→get_next
                 Adjust_Branches(n)
                 n=Temp
              else n = skip(n)
              Trfm = Trfm ∧ Tr
              break
          case J:
              return Trfm
     return Trfm }
```

Alg. 3.1. Algorithm for recursive transformation of a PFG.

In Alg. 1 the function Apply_Transformation() applies a set of transformations to alternate execution paths of a conditional. if the transformations are completely successful then it will yield a single execution path resulting from a conditional thus eliminating the particular conditional. In a case where alternate execution paths cannot be completely reduced it will minimize number of alternate execution paths. These transformations coupled with the code emitter will produce the target code for the transformed RTE-1 programs.

Given in Alg. 3.2 and Alg 3.3 are the functions Apply_Transformation() and Apply_Minimize() respectively. Apply_Transformation simply applies a set of transformations by considering branches pair-wise. Apply_Minimize applies the transformations to branches that are linear, again considering branches pair-wise.

```
boolean PFG::Apply_Transformation(Node *n) {
unsigned i,j;
    i=0;j=1;
    while (i < ((n->nb) - 1))
        apply_T(n,i,j);
        j++; if (j >= (n->nb)) { i++; j=i+1;}
if ((n->nb) == 1) return TRUE; else return FALSE; }
```

Alg. 3.2. Apply_Transformation().

```
boolean PFG::Apply_Minimize(Node *n) {
unsigned i,j,a,b;
    i=0;j=1;
    while (i < ((n− > nb) − 1))
        a=linear(n− >next[i]);
        b=linear(n− >next[j]);
        if (a && b)
            apply_T(n,i,j);
            j++; if (j >= (n− > nb)) { i++; j=i+1;}
        else
            if (!a && !b)
                i++;j=i+1;
            else
                if (!b)
                    j++;
                    if (j >= (n− > nb)) { i++;j=i+1;}
                else if (!a) {i++;j=i+1;}
if ((n− > nb) == 1) return TRUE; else return FALSE; }
```

Alg. 3.3. Apply_Minimize().

4. Determining the Efficiency of the Transformations

4.1 Definition of Efficiency

In order to determine the efficiency of the transformations it is necessary to apply the transformations to random PFGs and find the average efficiency. The efficiency (E) of a transformation can be defined as,

$$E = \frac{|PFG^O| - |PFG^T|}{|PFG^O|}, \text{ if } |PFG^O| > 1$$

where $|PFG^O|$ is the sum of the number of branches of each fork node in the original PFG before applying the transformations and $|PFG^T|$ is the sum of the number of branches of each fork node after applying the transformations to $|PFG^O|$. If PFG^O is linear then $|PFG^O| = 0$, i.e. there will be no fork nodes, thus the the number of branches as a result of fork nodes will be zero.

To evaluate the efficiency of the transformations it is necessary to randomly generate PFGs that model RTE-1 programs. In order to generate random PFGS the following model is defined. Let $G(M, d, b_f, f_p, s_p, c_p)$ be a function that generates a random PFG with the following properties.

Let M be a vector specifying the number of nodes (C,S,F) to be generated at depth i where $0 \le i \le d$.

$$M = \begin{bmatrix} m_0 \\ m_1 \\ \vdots \\ m_i \\ \vdots \\ m_d \end{bmatrix}$$

23

Also,

$$f_p + c_p + s_p = 1$$

where f_p, c_p and s_p gives the proportion of F,C and S nodes at depth i of the PFG being generated. Whenever a fork node is generated it contains on the average β branches representing alternate execution paths. The number of branches generated at a fork node is uniformally distributed on the values $2, 3, \cdots, b_f$, where b_f is the specified maximum number of branches to be generated.

Then β (the *branching factor* of the PFG) is given by,

$$\beta = \sum_{i=2}^{b_f} i/(b_f - 1) = (b_f + 2)/2.$$

Thus, by varying the parameters of G it is possible to generate a wide variety of PFGs. By applying the transformations to these random PFGs it is possible to calculate the average efficiency (E) of the transformations. The ideal set of transformations will give $E = 1$ for arbitrary PFGs. However in reality even transformations with $E = 0.5$ is a significant achievement.

4.2 The Results Obtained

It is necessary to determine how much of a reduction is possible on a PFG when the transformations are applied to a cluster representing a block of RTE-1 code that manipulate a related set of resources. Thus we only need to model blocks of RTE-1 code rather than whole programs. Also the efficiency is computed on blocks of code that manipulated a single resource.

Intuitively, notice that, whenever $c_p = 0$ or $s_p = 0$ then $E = 1$ since we can completely transform a PFG that consists only simple segments or a PFG that consists of only critical segments. For the graphs given in Fig. 4.1 to Fig. 4.6 the following values are chosen for the parameters M, b_f and d.

$$M = \begin{bmatrix} 1 \\ 12 \\ 8 \\ 6 \\ 4 \\ 4 \end{bmatrix}, \quad b_f = 3, \quad d = 6, \quad \beta = 2.5$$

The values for M (the total number of segments at depth i) are chosen such that it will closely resemble a block of RTE-1 code in terms of the number of instructions. Using the parameters f_p, c_p and s_p it is possible to vary the ratios of F, C and S segment nodes at each level of nesting. Thus, $f_p + c_p + s_p = 1$ define the space of all the programs that can be generated for fixed values of M, b_f and d. for the graphs given in fig. 4.1 and 4.2 the following ranges of values are chosen for c_p and s_p:

$$\left. \begin{array}{l} c_p = s \\ s_p = 1 - f_p - s \end{array} \right\} s = 0, \cdots, 1 - f_p \text{ step size} = (1 - f_p)/10.$$

where $f_p = 0.275, 0.35, 0.425, 0.5$. For each point in the space (f_p, c_p, s_p) a random PFG is generated and its efficiency is calculated. The average efficiency is computed

over 100 samples per point for the graph given in fig. 4.1. The graph in fig. 4.3 gives the probability that the transformation will have a efficiency grater than x $(P(E > x)$, $0 < x < 1)$, based on the distribution of efficiency given in fig. 4.2.

For the graphs in fig. 4.4 to 4.5 the following values are chosen for f_p, c_p, s_p.

$$\left.\begin{array}{l} c_p = (1 - f_p)/2 \\ s_p = (1 - f_p)/2 \end{array}\right\} 0 < f_p < 0.5.$$

The above values represent PFGs that have equal number of C and S segments. Thus, this particular case represent PFGs that resist transformations with a high probability. The probability that the efficiency is grater than x $(P(E > x), 0 < x < 1)$ is given in fig. 4.6. Form the graph in fig. 4.6 it can be clearly observed that in this case the probability that $E > 0.5$ is less than 20%. However, in the more general case (from fig. 4.3) the average efficiency is 0.4 with with a probability grater than 80%.
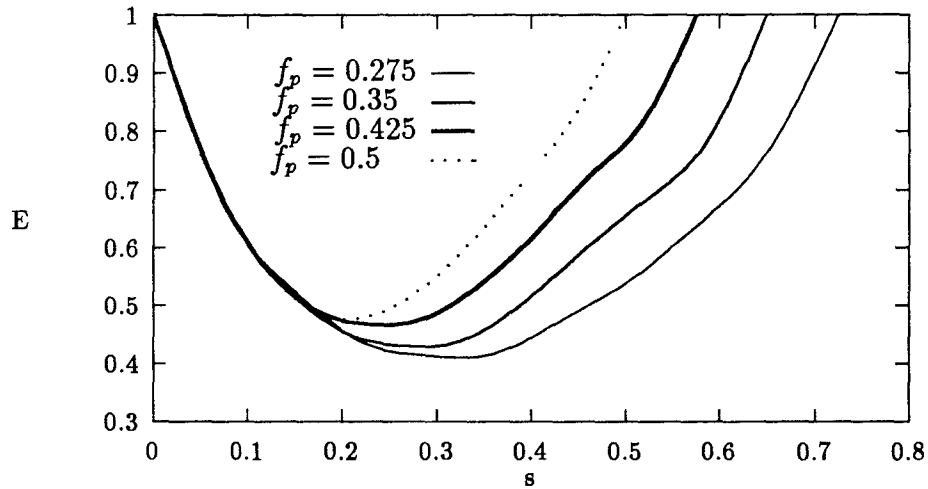
Fig. 4.1 Average efficiency of the transformations for the values $c_p = s$, $s_p = 1 - fp - s$, $0 \leq s \leq (1 - f_p)$, and Sample Size = 100.
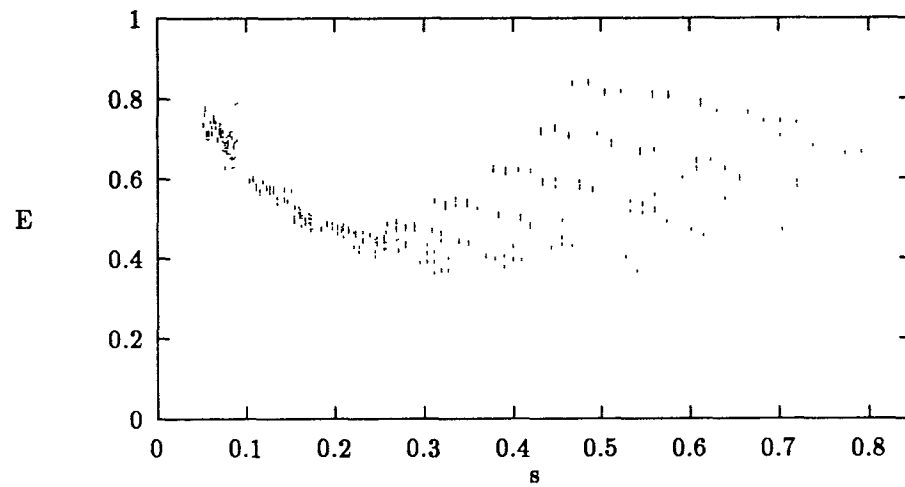
Fig 4.2 Scatter plot of E for $0 < f_p < 0.5$, $c_p = s, s_p = 1 - fp - s$ and $0 < s < (1 - f_p)$.
Sample Size $= 100$.
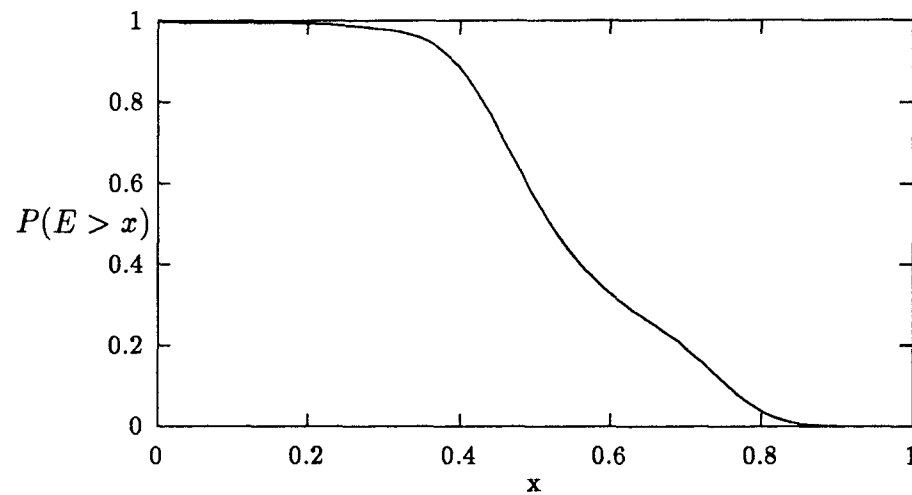
Fig 4.3 Probability distribution of $P(E > x)$ for the data given in Fig. 4.2
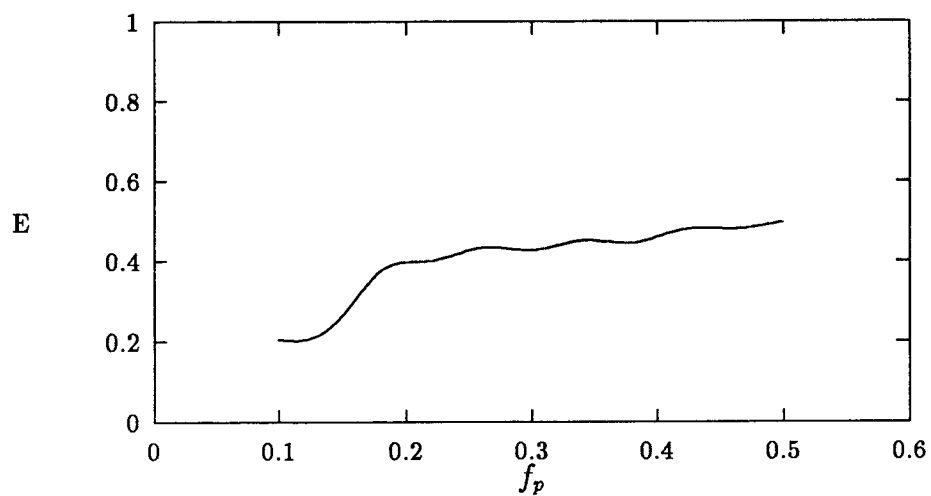
Fig 4.4 Average efficiency of the transformations for the values $c_p = (1 - f_p)/2$, $s_p = (1 - fp)/2$, and Sample Size $= 200$.
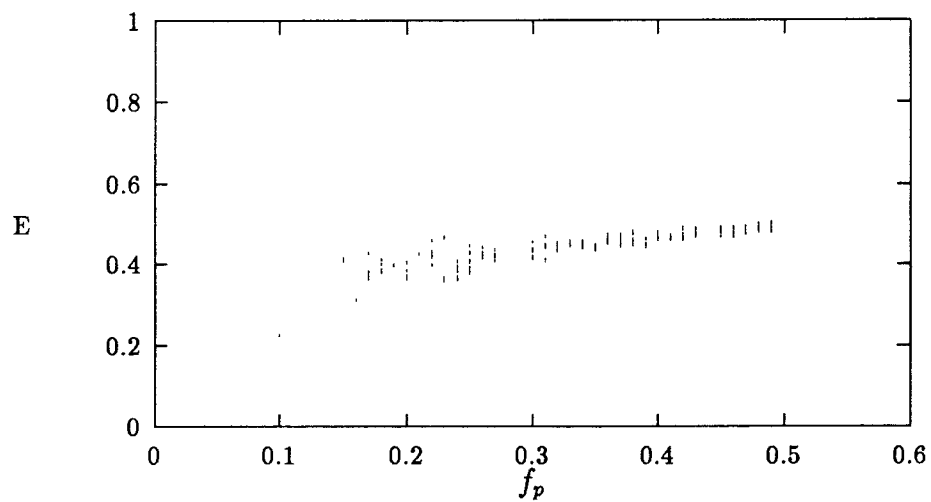
Fig 4.5 Scatter plot of E for $0 < f_p < 0.5$, $c_p = (1 - f_p)/2, s_p = (1 - fp)/2$. Sample Size $= 200$.
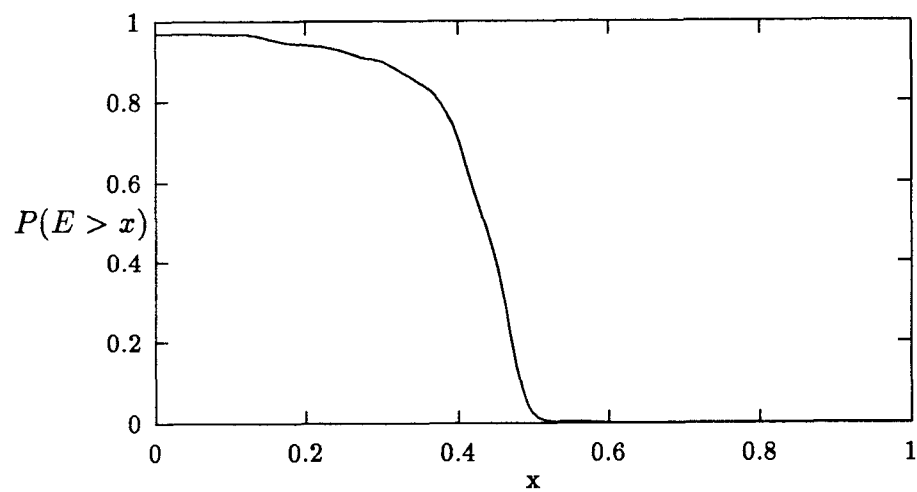
Fig 4.6 Probability distribution of $P(E > x)$ for the data given in fig. 4.5.

5. Acquisition of Resources in a Parallel Real Time Environment

Consider a set of processes written in RTE-1 that has explicit access to resources in the system. The task of the schedulability analyzer is to determine the execution times of each process in the system and provide a summary of the timing characteristics in a form that will inform the user whether or not his processes conform to the expected deadlines. Of course, in order for schedulability analysis to take place a static hardware configuration must be specified. In the rest of this paper the hardware model described in section 2.2 is assumed.

Objective of the transformations, described in section 3, is to reduce the problem size of the analysis algorithm that compute the static schedules of the processes. The analysis algorithm, described in this section, computes the static schedules for set of parallel processes which is used to determine the request time and the release time of each resource requests in each process. The processes are allowed to have nested conditionals. Since some processes may resist transformations in general the maximum level of nesting my be as much as in the original processes. As described in section 3, a set of PFGs are are created corresponding to the processes and the transformations are applied reducing the original PFGs as much as possible. The reduced PFGs are used to compute the release times for its associated resource requests. The static schedules of the reduced PFGs are used to determine the delays in order to pad the original processes appropriately such that it will conform to the precomputed static schedules. The original PFGs (i.e. the PFGs corresponding to the processes) will be needed to pad the original processes with the delays since transformations destroys the program flow structure of the original PFGs. The details of padding the original

processes will be explained in in section 5.2.

The following model is defined in order to express the details of resource acquisition in real time.

5.1 Restricted Resource Contention

The problem of exact schedulability analysis in the presence of shared resources contain an NP-complete problem. In order to get around this, one my consider the following techniques for timing analysis.

1. Employ deadline extending transformations,

2. Restrict the model of resource contention.

The second approach is considered in this paper.

Let n be the number of processes and m be the number of resources in the system. Also, let p_i be the ith process, R_j be the jth resource and lrt_j be the last release time of resource j with respect to the current time t in the system. Also, it is assumed that each processor is capable of maintaining the global time in its local system accurately.

In principle the queue size for a particular resource may range form 0 through $n - 1$. Thus the time taken to satisfy a request to a resource by a particular process may vary depending on the branches taken by the processes. Thus any analysis that needs to compute the release times of a resource will have to consider very combination of possible requests to that particular resource. Hence, the following restrictions are imposed in order to eliminate the combinatorial explosion that may take place in the analysis algorithm.

1. Whenever there is a queue of requests to a resource each process requesting the resource will have a static release time long enough to include any combination of requests that may occur in that interval.

2. Each process requesting a particular resource will be held until the static release time even if the process completes its request before that time.

Also, let $\Phi_{i,j,k,t}$ represent the event of process i requesting resource j for k units of time at time instance t. Let $\Psi_{i,j,\Delta,r}$ represent the event after accessing resource j by process i with a delay of Δ and being released at time r. Thus Ψ represent resolved resource requests. In general $r \geq t + \Delta + k + c$ and $r = t + \Delta + k + c$ iff the resource j was not busy at time $t + \Delta$, where c represent a communication delay. The extra delay Δ associated with a request will be explained later.

A resource request $\Phi_{i,j,k,t}$ is allowed to delay up to c units of time, i.e. if a request is dispatched at time instant t and that request is queued up at the resource request queue at time t' then $t < t' \leq t + c$, and $c > 0$. In a token ring network $c = Kn$ where n is the number of nodes in the network and K is a constant depending on the packet size and the capacity of the communication channel.

Consider p resource requests to the same resource R_j. Let t_i represent the time of the resource request and u_i represent the amount of time the resource R_j is used by request i. Also, assume that $t_0 < t_1 < \cdots < t_{p-1}$, i.e. sorted by the request time and $t_0 + \sum_{j=0}^{i} u_j > t_{i+1}$, for $i = 0, 1, \cdots, p - 2$, i.e. the requests dispatched at $t_0, t_1, \cdots, t_{p-1}$ will form a queue for the resource R_j. Then the resource busy interval R_j^{int} of R_j is $< t_0, t_0 + \sum_{i=0}^{p-1} u_i + c >$, which states that from the time instant t_0 this resource will be busy for $\sum_{i=0}^{p-1} u_i + c$ units of time satisfying the p requests.

Thus, we compute the release time of resource R_j, with respect to the p resource

requests, to be $r = t_0 + \sum_{i=0}^{p-1} u_i + c$. Hence, all the processes associated with resource busy interval R_j^{int} will be released at time instant r. Next, the algorithm that compute the release times in the above described manner is presented.

The algorithm 5.1 computes the release times and the delays associated with a resource request. Also it updates the PFG with the computed values. The input to the algorithm is the communication delay c and a set of PFGs. The algorithm considers a row of requests (R) comprising of the resource requests of all PFGs in order to determine the earliest resource request interval. The row of requests are obtained by a *tree of pointers* that point to critical segments in the PFG. For each iteration the algorithm finds the set of requests R'' that fall in the earliest resource busy interval with respect to resource j and computes the release time for this resource busy interval. Then the requests in R'' are *advanced* to the next set of requests. When $R = \emptyset$ the algorithm terminates.

If a resource request at time t is before the last release time lrt_j for resource j (i.e. $t < lrt_j$) then Δ units of time are delayed before making the request to resource j, where $\Delta = lrt_j - t$. For example consider the two PFGs given in fig 5.1. According to Alg. 5.1 the initial set of resource request are :

$R = \{\Phi_{1,1,8,3}, \Phi_{2,2,2,1}\}$

and after resolving the request, $R'' = \{\Psi_{2,2,0,3}\}, lrt_2 = 3$.

i.e the resource request R_2 has a release time of 3 time units and $\Delta = 0$.

After advancing resource request R_2 the row R becomes:

$R = \{\Phi_{1,1,8,3}, \Phi_{2,3,3,4}\}$

and $R'' = \{\Psi_{1,1,0,11}\}, lrt_1 = 11$,
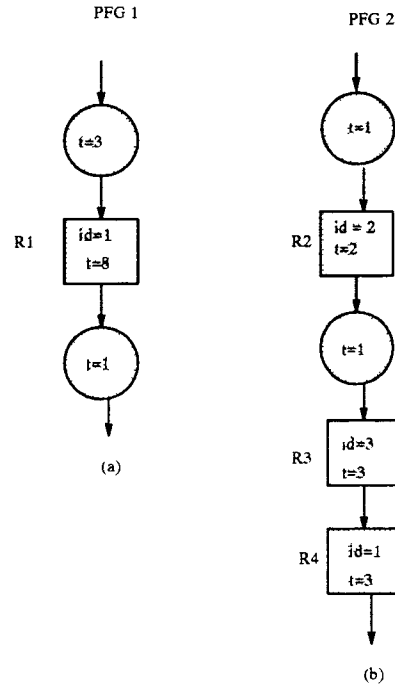
Fig. 5.1. Example of two linear PFGs.

i.e the resource request R_1 has release time of 11 time units and $\Delta = 0$,

and after advancing request R_1 the row R becomes:

$$R = \{\Phi_{2,3,3,4}\}$$

and $R'' = \{\Psi_{2,3,0,7}\}, lrt_3 = 7$.

After advancing R_3 the row R becomes:

$$R = \{\Phi_{2,1,3,7}\}$$

at this point we cannot let this request in to system since $t < lrt_1$, thus it is

necessary to delay this request by $\Delta = lrt_1 - t = 4$ units of time.

and $R'' = \{\Psi_{2,1,4,14}\}, lrt_1 = 14$.

Generating the resource request R_4 in PFG 2 (given in fig. 5.1 (b)) at time 7 will

Compute_Release_Times(c,PFGs)
$R, R', R'' \in \{\Phi^*, \Psi^*\};\ \mathrm{lrt}_j = 0$
$R =$ initial row of resource requests of N PFGs.
$c =$ Communication Overhead
WHILE $(R \neq \emptyset)$
   $\Phi_{*,j,*,*} = \min_t[R]$
   $R' =$ all $\Phi_{*,j,*,*} \in R$
   $R' = \Phi_{i,j,k,t} + R'$
   $R = R - R'.$
   SORT $R'.$
   $(r_t, R', R'') = $ Compute_Release_Times1(R', lrt_j, c)
   FOR $l = 0, \cdots, |R''| - 1$
      $\Psi_{i,j,\Delta,*} = Head(R'')$
      $R'' = R'' - \Psi_{i,j,\Delta,*}$
      $R'' = R'' + \Psi_{i,j,\Delta,r_t+c}$
   $lrt_j = \max(lrt_j, r_t + c)$
   Update_PFG(R'')
   $R'' = \mathrm{Advance}(R'')$
   $R = R + R'' + R'''$
END Compute_Release_Times

Alg. 5.1. Algorithm for computing release times.

cause the guarantee, of resource request R_1 to be released at time 11, to be violated.

Thus the request at R_4 is delayed by 4 time units. Note that in the above example

the communication delay is not considered.

$\text{Compute\_Release\_Times1}(R, lrt_j, c)$

$\quad R'' = \emptyset$

$\quad \Phi_{i,j,k,t} = \text{Head}(R)$

$\quad \text{IF } (lrt_j > t)$

$\qquad \Delta + = lrt_j - t$

$\qquad t + = lrt_j - t$

$\quad R'' = R'' + \Psi_{i,j,\Delta,*}$

$\quad R = R - \Phi_{i,j,k,t}$

$\quad r_t = t + k$

$\quad \text{flag} = \text{TRUE}$

$\quad \text{FOR } l = 0, \cdots, |R| - 1 \wedge flag$

$\qquad \text{flag} = \text{FALSE}$

$\qquad \Phi_{i,j,k,t} = \text{Head}(R)$

$\qquad \text{IF } (lrt_j > t)$

$\qquad\quad \Delta + = lrt_j - t$

$\qquad\quad t = lrt_j - t$

$\qquad \text{IF } (r_t < t)$

$\qquad\quad \text{flag} = \text{TRUE}$

$\qquad\quad r_t + = k$

$\qquad\quad R'' = R'' + \Psi_{i,j,\Delta,*}$

$\qquad\quad R = R - \Phi_{i,j,k,t}$

$\quad \text{RETURN } (r_t, R, R'')$

$\text{END Compute\_Release\_Times1}$

Alg. 5.2. Algorithm that compute the resource busy interval.

Tree Of Pointers:

A tree of pointers (TOP) is a rooted k-ary tree. The purpose of a TOP structure is to represent a list of resource requests that may occur at a given time instant. A separate TOP structure is maintained for each PFG that needs to be analyzed. A node of a TOP structure consists of a time t and a pointer to a critical segment node p_c corresponding to a resource request. In general, for PFG, a tree of pointers T_i is maintained. The leaf nodes of T_i represent the current set of resource requests of PFG$_i$. The row of current resource requests (R) are formed by taking all the leaf nodes in each T_i ($i = 0, \cdots, n-1$). For example fig. 5.3 (a) shows the TOP for the PFG given in fig. 5.2 with its leaf nodes pointing to the initial set of resource requests. Fig 5.3 (b) shows the TOP after advancing the leaf node l_2, i.e. after resolving resource request R_2 of PFG in fig. 5.2. The release time of resource request R_2 is 12 (assuming $c = 1$). Fig 5.4 (d) shows the TOP after advancing the leaf node l_7. When l_7 is advanced (with a release time of 17), going across a simple segment node of 1 unit, a join node is encountered, which causes the time 18 of node l_7 to propagate to its parent, this situation is show in fig. 5.5 (d). Similarly, advancing l_{10} gives the TOP in fig. 5.4 (e). When algorithm 5.1 terminates it results in a TOP with a single node giving the termination time of that PFG (see fig. 5.4 (f)).

The following steps are take when advancing a *node* of a TOP.

1. *node.t* = release time of the resource request pointed by *node*.

2. Advance to the next node in the PFG, i.e. *node.p_c* = *node.p_c* $\rightarrow$ *next*, if the next node is a simple segment node then add its time to *node.t* and apply steps 2-4 else found next request, return.

3. If *node.p_c* is a fork node then create a child for each branch in the fork node

and set the time t of each new child to its parents current time. Apply steps 2-4 recursively to each child until a critical segment node is reached or end of the PFG reached.

4. If $node.p_c$ is a join node then if $node.t$ is grater than the time of parent then propagate the time $node.t$ to its *parent*. Delete *node.* if *parent* becomes a leaf node apply steps 2-4 to the *parentnode.*

Complexity of algorithm 5.1.

A worst case scenario for the algorithm is when for each iteration of the while loop $|R'| = 1$. The search for the minimum element in R is $\bigcirc(n\beta^d)$. The while loop will execute $\bigcirc(Dn\beta^d)$ since we need to resolve all the requests in a row, where D is the maximum number of rows that can be constructed as a result of advancing resource requests, i.e. the maximum number of critical segments in any path in the DAG. Thus the total worst case complexity is $\bigcirc(Dn^2\beta^{2d})$. A lower bound for the algorithm is $\Omega(Dn^2)$. This lower bound is achieved when all the PFGs to be analyzed are linear.
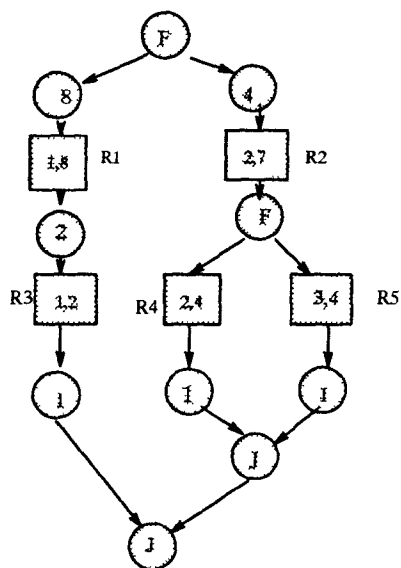
Fig. 5.2.
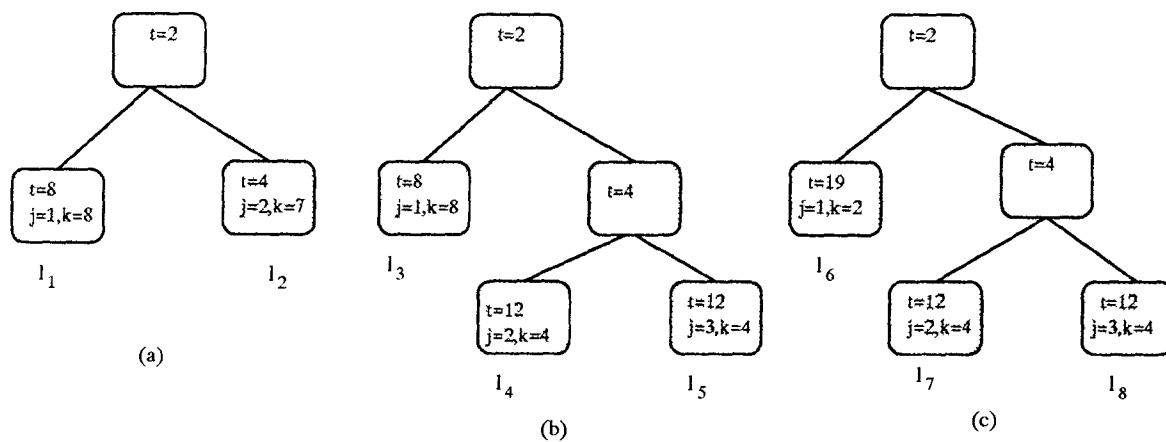
Fig. 5.3. (a) Initial list of requests pointed by the leaf nodes of each tree for the PFGs given in Fig. 5.2. (b) After advancing request R2. (c) After advancing R1.
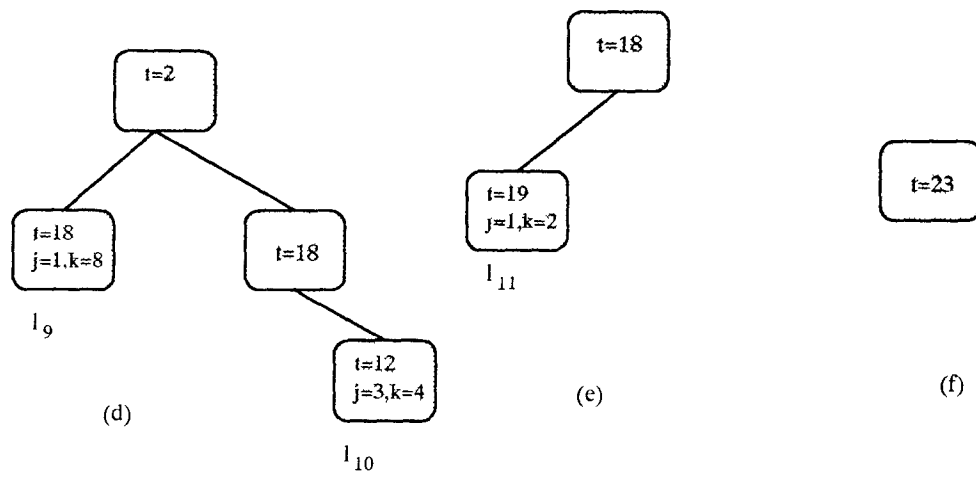
Fig. 5.4 (d) After advancing request R4. (e) After advancing request R5. (f) After advancing R3.
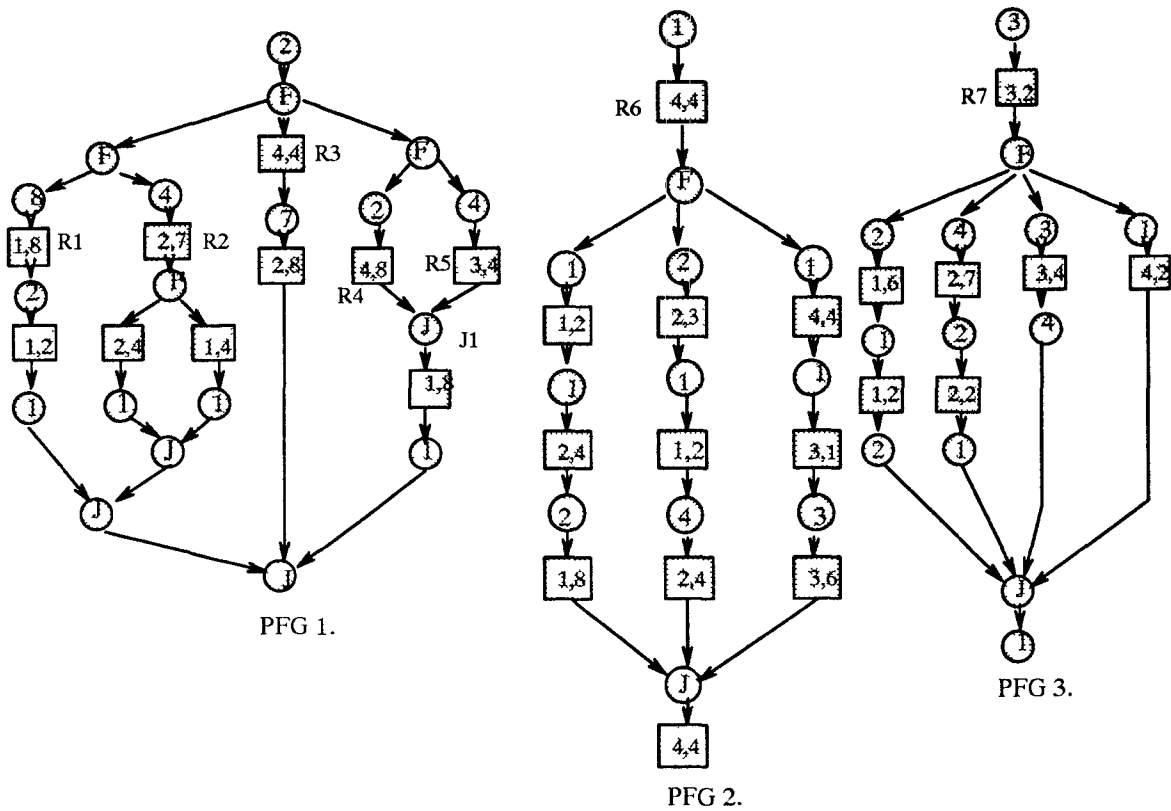
Fig. 5.5. Three PFGs corresponding to three processes that neeed to be analyzed.

As an example the algorithm 5.1 is applied to the PFGs given in fig. 5.6 resulting in the static schedules given in fig. 5.7. The slow down of a process is calculated with respect to the termination time of the same PFGs without any contention for resources, i.e. the the slow down can be defined as,

$$\text{Slow Down} = \frac{\text{termination time of a process with contention}}{\text{termination time of a process without contention}}$$

The slow down of a process gives an idea about the loss of parallelism due to shared resource contention.

PFG1	PFG2	PFG3

```
PFG1:
( S ,2)
( F ,0) {
  [
    ( F ,0) {
      [
        ( S ,8)
        ( C ,8,1)
        ( S ,2)
        ( C ,2,1)
        ( S ,1)
      ][
        ( S ,4)
        ( C ,7,2)
        ( F ,0) {
          [
            ( C ,4,2)
            ( S ,1)
          ][
            ( C ,4,1)
            ( S ,1)
          ]
        }
        ( J ,0)
      ]
    }
    ( J ,0)
  ][
    ( C ,4,4)
    ( S ,7)
    ( C ,8,2)
  ][
    ( F ,0) {
      [
        ( S ,2)
        ( C ,8,4)
      ][
        ( S ,4)
        ( C ,4,3)
      ]
    }
    ( J ,0)
    ( C ,8,1)
  ]
} ( J ,0)

PFG2:
( S ,1)
( C ,4,4)
( F ,0) {
  [
    ( S ,1)
    ( C ,2,1)
    ( S ,1)
    ( C ,4,2)
    ( S ,2)
    ( C ,8,1)
  ]
  [
    ( S ,2)
    ( C ,3,2)
    ( S ,1)
    ( C ,2,1)
    ( S ,4)
    ( C ,4,2)
  ]
  [
    ( S ,1)
    ( C ,4,4)
    ( S ,1)
    ( C ,1,3)
    ( S ,3)
    ( C ,6,3)
  ]
}
( J ,0)
( C ,4,4)

PFG3:
( S ,3)
( C ,2,3)
( F ,0) {
  [
    ( S ,2)
    ( C ,6,1)
    ( S ,1)
    ( C ,2,1)
    ( S ,2)
  ]
  [
    ( S ,4)
    ( C ,7,2)
    ( S ,2)
    ( C ,2,2)
    ( S ,1)
  ]
  [
    ( S ,3)
    ( C ,4,3)
    ( S ,4)
  ]
  [
    ( S ,1)
    ( C ,2,4)
  ]
}
( J ,0)
( S ,1)
```

Fig 5.6. The three PFGs given in fig. 5.5 represented according to the grammar given in fig. 3.1.

PFG1	PFG2	PFG3

(S ,2)
(F ,0) {
  [
  (F ,0) {
    [
    (S ,8)
    (C ,8,1,41,0)
    (S ,2)
    (C ,2,1,51,0)
    (S ,1)
    ][
    (S ,4)
    (C ,7,2,23,0)
    (F ,0) {
      [
      (C ,4,2,43,0)
      (S ,1)
      ][
      (C ,4,1,41,0)
      (S ,1)
      ]
    }
    (J ,44)
    ]
  }
  (J ,52)
  ][
  (C ,4,4,20,0)
  (S ,7)
  (C ,8,2,43,0)
  ][
  (F ,0) {
    [
    (S ,2)
    (C ,8,4,20,0)
    ][
    (S ,4)
    (C ,4,3,19,2)
    ]
  }
  (J ,20)
  (C ,8,1,41,0)
  ]
}
(J ,52)
Termination time= 52
Slow Down   = 1.79
	(S ,1)
	(C ,4,4,20,0)
	(F ,0) {
	  [
	  (S ,1)
	  (C ,2,1,41,0)
	  (S ,1)
	  (C ,4,2,50,1)
	  (S ,2)
	  (C ,8,1,63,0)
	  ]
	  [
	  (S ,2)
	  (C ,3,2,43,1)
	  (S ,1)
	  (C ,2,1,51,0)
	  (S ,4)
	  (C ,4,2,62,0)
	  ]
	  [
	  (S ,1)
	  (C ,4,4,29,0)
	  (S ,1)
	  (C ,1,3,34,0)
	  (S ,3)
	  (C ,6,3,46,0)
	  ]
	}
	(J ,63)
	(C ,4,4,70,0)
	
	Termination time= 70
	Slow Down = 1.67
		(S ,3)
		(C ,2,3,8,0)
		(F ,0) {
		  [
		  (S ,2)
		  (C ,6,1,41,0)
		  (S ,1)
		  (C ,2,1,51,0)
		  (S ,2)
		  ]
		  [
		  (S ,4)
		  (C ,7,2,23,0)
		  (S ,2)
		  (C ,2,2,43,0)
		  (S ,1)
		  ]
		  [
		  (S ,3)
		  (C ,4,3,19,0)
		  (S ,4)
		  ]
		  [
		  (S ,1)
		  (C ,2,4,29,11)
		  ]
		}
		(J ,53)
		(S ,1)
		
		Termination time= 54
		Slow Down = 1.74

Fig 5.7. The corresponding static schedules for the PFGs given in fig. 5.6. Algorithm 5.1 is used for the analysis. The critical segment nodes have the following fields (C,t,id,r_t,Δ) and $c = 3$.

Resource Request Interrupt Handler:
$$\Phi_{*,Release\_Time} = \text{Get\_Request}();$$
Dispatch_Request(Φ_*);
Sleep Until *Sys_time = Release_Time* ;
Return;

Fig 5.8. Example of a runtime abstraction for handling resource requests.

5.2 Transforming the Processes in Order to Conform to the Release Times

Once the release times are computed for each request it is necessary to ensure that at runtime the processes conform to the timing characteristics obtained by the analysis. The runtime system will simply dispatch a request to a resource across a network or simply queue it up at the local processing element. Further more the runtime system will be informed by the resource when the access is completed. Thus it is the task of the runtime system to delay the process requesting the resource until the precomputed release times. In order to achieve this each resource request will accompany the precomputed release time so that the runtime system can delay until the release time is reached by the local system clock.

Delaying a process after a resource request in it self is not enough to maintain the desired timing characteristics. It is also necessary at a join node to delay as long as the longest branch. This can be be achieved by simply adding a delay at the join nodes that corresponds to the maximum time that may be taken by the corresponding execution paths. Thus the process will idle at a join until the desired amount of time has passed. At runtime, when the system encounters a resource request the system calls a service routine that will handle the request. An example of a runtime abstraction for handling a resource request is given in fig. 5.8.

6. Conclusion

The primary objective of this thesis was to, (1) extend the language in order to allow for nested conditionals, (2) include the facility to represent multiple resources in the language, and extend the transformations and the analysis algorithms to take account of the above changes; which is successfully accomplished. In section 3 it is shown how the transformations are extended to handle nested conditionals. In section 4 the efficiency of the transformations are defined and applied to a particular case. In section 4.2 the statistics of the results of the transformations are given, showing clearly that in certain cases the the transformations are capable of reducing the problem size by 40% with a probability of 80%. The transformations and the analysis algorithm are available as set of library routines in C++, which can be used as a testbed for constructing and testing various types of transformations and heuristics for analysis. Extensions and improvements are given in section 6.1.

6.1 Future Extensions and Improvements

The work done in this thesis provides a comprehensive basis for building a complete compiler/schedulability analyzer for a supper set of the real time language that is given in fig. 2.1. However, more work is needed to determine the accuracy of the static schedules and the slow down caused as a result of the heuristics. Also, it is possible to introduce more type B transformations. The following is a list of possible extensions.

1. It is possible to introduce more type B transformations. Also, the cost of the transformations must be determined with respect to the reduction in the cost of schedulability analysis.

2. When deadline extending (Type C) transformations are used it is important to determine how much of a deadline extension is caused. Such an estimate will give an idea of whether using type C transformations are in fact feasible or not.

3. The slow down of a process due to heuristics of the analysis algorithm must be computed. This problem it self is NP-complete thus an approximate calculation of the slow down will suffice.

6.2 A Generic Real Time System: A Proposal

All real time systems are application specific in terms of the hardware requirements and access a unique set of resources. Thus, building a general purpose real time hardware architecture is a serious failing. Instead, the real time system designers should posses a suit of tools that allow the specification of a hardware configuration that exactly match the application requirements. Such a design tool should have a specification tool for defining a hardware configuration, a compiler/schedulability analyzer and a simulation tool. The specification tool will allow the designers to express a particular hardware configuration. The compiler/schedulability analyzer will produce the transformed object code for a set of processes, executable on the hardware configuration specified by the specification tool. The simulator will allow the designers to simulate the set of real time processes and fine tune them. Once the simulation tool provide satisfactory evidence of timelyness of the processes the hardware specification is implemented. This activity is shown in fig. 6.1. The hardware specifications are derived from a collection of, heterogeneous processing elements, interfaces to the environment and other facilities such as shared address spaces, interrupt handlers and communication network topologies. Fig 6.1 Shows the activities in the proposed system for designing a real time system.
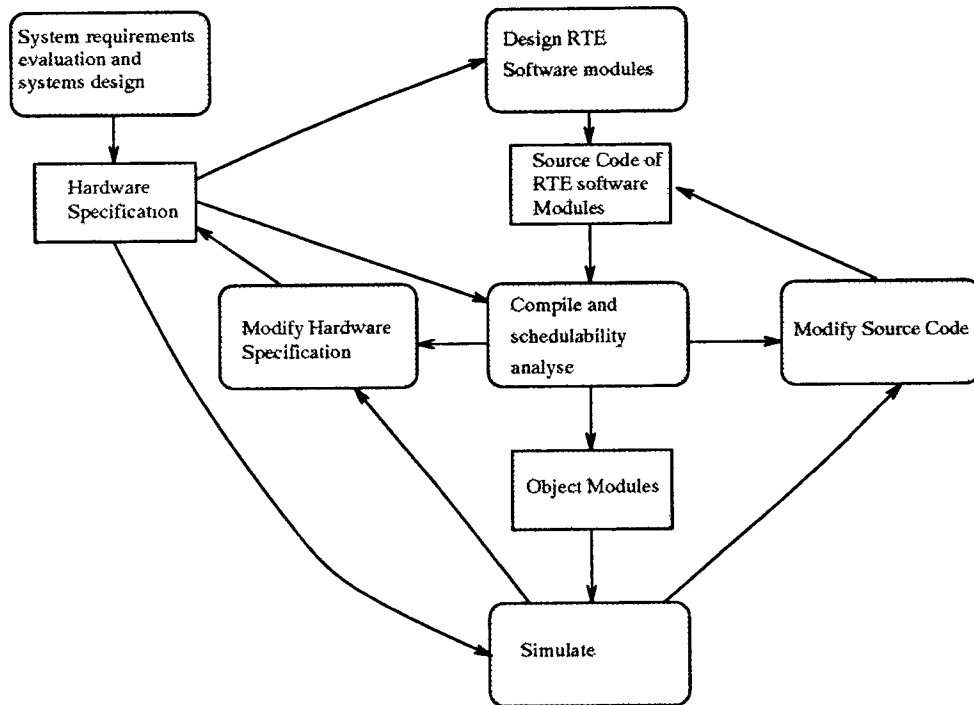
Fig. 6.1. System Development Cycle.

The main issue in hard real time computing is not whether all processors are fully utilized, but simply deadline satisfaction. Thus, under-utilization of some or even all processors are acceptable form the standpoint of *deadline satisfaction, predictability and dependability.* Adaptability versus redundance poses a considerable challenge to any designer. However, this is a issue that is solely in the realm of system design. Thus the real time system design tools must provide sufficient degrees of freedom to the implementors.

The processing elements (PE) are simple components, consisting of a local memory and communication ports to a network, without any caches or virtual memory. Consider a collection of simple hardware entities such as PEs, network topologies(NT), DAC,ADC, and software modules that may be used in a hardware specification for a real time architecture. A *resource* in such a specification may identify a simple entity or a complex entity. A complex entity is a collection of simple entities. Fig. 6.2 shows a configuration of a system with three PEs, five resources and a token bus NT. The relative usage-dependence specified between two entities will identify a particular resource and the consumers (users) of that particular resource. Given the specification of the architecture in fig. 6.2 it is possible to schedulability analyze and compile a set of processes, producing transformed real time code that will execute in a timely manner.

The low cost, of processing elements and local area networks, makes the above described methodology for constructing a hard real time system feasible. However, the proposed methodology does not provide a quick fix to todays real time requirements in industry, since to build such a tool will require an significant amount of time and money. Let us consider in detail the structure of a specification for a hardware architecture. Fig. 6.3 shows the specification of the hardware architecture given in
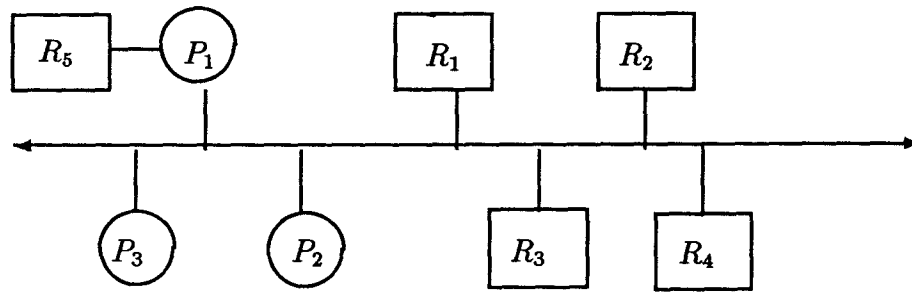
Fig. 6.2. An Instance of a hardware architecture.

P_1 Uses R_1, R_2, R_3, R_4, R_5.
P_2 Uses R_1, R_2, R_3, R_4.
P_3 Uses R_1, R_2, R_3, R_4.
$P_1, P_2, R_1, R_2, R_3, R_4 \in NT_1$.
NT_1 Is a Token Bus NT.
R_1 Is a CPU.
P_1, P_2, P_3 IS a CPU.
R_2 Is a SM.
R_3 Is a Bridge.
R_4 Is a ADC.
R_5 Is a DAC.

Fig. 6.3. Specification for the h/w architecture in fig. 6.2.

fig. 6.2. The key words *Uses,In*, and *Is a* are used to identify the relation ships between the entities and types of the entities respectively. Such a specification can be used to derive the required components within a entity, i.e. connection to a network may or may not be available for a particular resource, as in the case of R_5 in Fig. 6.2. Also the specification is used to determine the overhead cost of communication and other information regarding the timing characteristics of the components.

Appendix 1: Library Reference

NAME

PFG - Program Flow Graph.

SYNOPSIS

#include <graph.h>

```
class PFG {
public:
    PFG();
    PFG(char *infile);
    PFG(char *infile, ofstream *out);
    Node *get_start();
    void print_PFG(Node *node); // Write this PFG to file
    Node *read_PFG();           // Read a PFG from file
    Node *generate(unsigned M[],unsigned l, unsigned bf,
            float fp,float cp, float sp);
    void cluster_PFG(Node *node);
    unsigned transform(Node *node);
    double N_branches(Node *node);
}
```

DESCRIPTION

The class PFG provides methods for manipulating a DAG which represent the timing and program flow characteristics of a RTE-1 programs.

A PFG can be declared as:

PFG G;

PFG H(in);

PFG I(in,out);

Which declares G to be a empty DAG, H to have an input file *in* and I to have an input file *in* and a output file *out*. The input file can be used to read in a DAG

representing a RTE-1 program, and can be written out to a file specified by *out*. The external format for a PFG is given by the grammar in fig. 3.2.

PFG G(in,out);

G.read_PFG();

G.cluster_PFG();

G.transform(G.start);

G.print_PFG(G.start);

Which declares *G* and reads a PFG form the input file specified by *in*. Preclusters the PFG in order to reduce sequences of simple segments. Then, applies the transformations and writes the transformed PFG to the file specified by *out*. The class PFG and its associated member functions are given in appendix 2, from page 55 to page 83.

NAME

PFGS - Program Flow Graphs. This class models a collection of program flow graphs.

SYNOPSIS

```
#include <list.h>
class Traversal;


class PFGS {
public:
    Traversal *T[MAX_PFGS]
    unsigned np,x;
    PFGS(unsigned n, char *arg[], ofstream *out);
    TreeNode *row(unsigned i, unsigned j);
    Node *node(unsigned i, unsigned j);
    void advance(unsigned i, unsigned j);
    void release_time_static(unsigned x);
    void release_time(unsigned x);
}
```

DESCRIPTION

The class PFGS provides methods for manipulating TOP structures associated with each PFG. The details of TOP structures are given in section 5.1. The class PFGS is instantiated with n PFGs which represent a set of parallel real time programs to be analyzed. For each PFG stored in the class PFGS a unique TOP structure is maintained where $T[i]$ points to the TOP structure class for the ith PFG. The TOP structure is realized by the class Traversal. The classes Traversal and PFGS and its associated member functions are given in appendix 2, from page 87 to page 96. The member function row(i,j) in class PFGS represent the jth resource request of the ith PFG represented in that class. Also, the member function advance(i,j) advances the jth resource request in the ith PFG. The details of advancing resource requests are given in section 5.1. The function compute_release_times(x) computes the static release times and the process termination times of n PFGs represented by the class PFGS. A communication delay of x units of time is allowed for when computing the release times. Also, compute_release_times(x) is the implementation of the algorithm 5.1 given in section 5.1.

DISCUSSION

The classes PFG, PFGS, and Traversal are used to compute the efficiency of the of the transformations given in section 4.2. The data for the graphs given in fig. 4.1 to fig. 4.6 are obtained by the programs *epoch.c, epoch1.c, epoch3.c* and *epoch4.c*, which are given in appendix 2, form page 101 to page 110. Also, the programs *release_times.c* and *transform.c*, given in appendix 2, form page 111 to page 114, are used to compute the static schedule given in fig. 5.7. By using the C++ member functions of the classes described above it is possible to implement different heuristics for analysis.

Appendix 2: Program Listings

```
#include "node.h"
unsigned lc=0;
static gps =0;
char str[100];
/*
    Global variables used for formatting the output.
    lc = Line Count,
    gps = indentation value.
    str = temporary string translation space.
*/


#define MAX_DEPTH 10
/*
    MAX_DEPTH defines the maximum depth allowed in a PFG.
    This is used in generating random PFGs by generate().
    The transformations and other general functions are not
    dependent on MAX_DEPTH.
*/

class PFG {
    void node_S_C(node_type nt,unsigned t,Node  **new_node,
        Node  **n,char &ch);
    void node_F(node_type nt,unsigned t,Node **new_node,
        Node **n,char &ch);
    char skip();
    node_type get_type();
    unsigned get_time();
    void exit_eof(char *s="");
    void get_id(unsigned &id);
    double n_branches;
    double n_branches_c;

    public:
    Node *start;
    ifstream *in;
    ofstream *out;
    double max(double a,double b) { if (a>b) return a; else return b;}
    char infile[25];
    PFG(char *,ofstream *);
    PFG(ofstream *);
    PFG(void);
    ~PFG();
```

```
        void add_node(Node *node,Node *at); /* add node at */
        Node *get_start() { return start;}
        void printf(Node *node);
        void print(char *str);
        void calc_depth(Node *node,unsigned d);
        Node *generate(unsigned M□,unsigned l,unsigned bf,
            float pe,float fp,float cp, float sp,Node *n,
            unsigned d,unsigned p_o=FALSE);
        Node *read_PFG(Node *n); /* read PFG form file: see PFG syntax */
        Node *cluster_sequence(Node *n);
        void cluster_PFG(Node *n);
        Node *adjust_branches(Node *n, Node *prev);
        unsigned transform(Node *n, Node *prev=NULL);
        unsigned apply_transformation(Node *n);
        void compute_lengths(Node *p,unsigned &l,unsigned &c_l,unsigned &s_l);
        void compute_times(Node *p,unsigned &t,unsigned &c_t,unsigned &s_t);
        void apply_T1(Node *p,unsigned &i,unsigned &j);
        void apply_T2(Node *p,unsigned &i,unsigned &j);
        void apply_T3(Node *p,unsigned &i,unsigned &j,unsigned t0,
                unsigned t1,unsigned c_l);
        void apply_T4(Node *p,unsigned &i,unsigned &j);
        unsigned  apply_T(Node *n,unsigned &r, unsigned &s);

        void apply_T6(Node *n,unsigned &i,unsigned &j,unsigned t0,
                unsigned t1);
        void test(Node *n);
        double compute_paths(Node *n);
        double max_depth(Node *n);
        double N_f();
        void n_f(Node *n);
        int linear(Node *n);
        unsigned apply_minimize(Node *n);
};

void PFG::print(char *str)
{
    *out<<"\n"<<str<<"\n"<<flush;
}

/*
    Computes the number of segments in total, number of
    critical segments and the number of simple segments
    in a branch where the pointer p points to the first
    segment in that particular branch.
*/
```

```
void PFG::compute_lengths(Node *p,unsigned &l,unsigned &c_l,unsigned &s_l)
{
    c_l=s_l=0;
    while (p->Type != J) {
        switch (p->Type) {
        case S: s_l += 1;
            break;
        case C: c_l += 1;
            break;
        default: error("Invalid type: compute lengths ");
            exit(0);
        }
        p =p->next[0];
    }
    l = c_l+s_l;
}


/*
    computes the total and sub times for a branch. similar to
    compute_length(). Only applicable to a brach of a conditional.
*/
void PFG::compute_times(Node *p,unsigned &t,unsigned &c_t,unsigned &s_t)
{
    c_t=s_t=0;
    while (p->Type != J) {
        switch (p->Type) {
        case S: s_t += p->t;
            break;
        case C: c_t += p->t;
            break;
        default: error("Invalid type: compute times ");
            exit(0);
        }
        p =p->next[0];
    }
    t = c_t+s_t;
}

/* The transformations T1 to T4 are equivalent to Tran_T() */

void PFG::apply_T1(Node *p,unsigned &i,unsigned &j)
{
unsigned k;
    if ((p->next[i])->Type == J){
        (p->next[i])->rc -= 1;
```

```
        for (k=i;k<(p->nb-1);k++)
            p->next[k] = p->next[k+1];
    } else {
        (p->next[j])->rc -= 1;
        for (k=j;k<(p->nb-1);k++)
            p->next[k] = p->next[k+1];
    }
    j=j-1;
    p->nb = p->nb - 1;
}


void PFG::apply_T2(Node *p,unsigned &i,unsigned & j)
{
unsigned k;

    if (((p->next[i])->t) > ((p->next[j])->t)) {
        delete p->next[j];
        for (k=j;k<(p->nb-1);k++)
            p->next[k] = p->next[k+1];
    }
    else {
        delete p->next[i];
        for (k=i;k<(p->nb-1);k++)
            p->next[k] = p->next[k+1];
    }
    j=j-1;
    p->nb = p->nb - 1;
}
void PFG::apply_T3(Node *p,unsigned &i,unsigned &j,unsigned t0,
            unsigned t1,unsigned c_1)
{
unsigned k;
    if (c_1==0) {
        if (t0<=t1){
            delete p->next[i];
            for (k=i;k<(p->nb-1);k++)
                p->next[k] = p->next[k+1];
            j=j-1;
            p->nb = p->nb - 1;
        } else {
            delete p->next[i];
            for (k=i;k<(p->nb-1);k++)
                p->next[k] = p->next[k+1];
            j=j-1;
            p->nb = p->nb - 1;
```

```
                        (p->next[j])->t += t0 - t1;
                }

        } else {
            if (t1<=t0) {
                delete p->next[j];
                for (k=j;k<(p->nb-1);k++)
                    p->next[k] = p->next[k+1];
                j=j-1;
                p->nb = p->nb - 1;
            } else {
                delete p->next[j];
                for (k=j;k<(p->nb-1);k++)
                    p->next[k] = p->next[k+1];
                j=j-1;
                p->nb = p->nb - 1;
                (p->next[i])->t += t1 - t0;
            }
        }
}

void PFG::apply_T6(Node *p,unsigned &i,unsigned &j,unsigned t0,
            unsigned t1)
{
unsigned k;
    if (t0<=t1){
        delete p->next[i];
        for (k=i;k<(p->nb-1);k++)
            p->next[k] = p->next[k+1];
        j=j-1;
        p->nb = p->nb - 1;
    } else {
        delete p->next[j];
        for (k=j;k<(p->nb-1);k++)
            p->next[k] = p->next[k+1];
        j=j-1;
        p->nb = p->nb - 1;
    }
}

void PFG::apply_T4(Node *p,unsigned &i,unsigned &j)
{
unsigned T;
unsigned k;
Node *ni;
```

```
Node *nj;
    ni = p->next[i];
    nj = p->next[j];
    T=TRUE;
    while (ni->Type !=J) {
        if (ni->Type != nj->Type){
            T=FALSE;
            break;
        }
        else if (ni->t<nj->t){
            T=FALSE;
            break;
        }
        ni = ni->next[0];
        nj = nj->next[0];
    }
    if (T) {
        delete p->next[j];
        for (k=j;k<(p->nb-1);k++)
            p->next[k] = p->next[k+1];
        j=j-1;
        p->nb = p->nb - 1;
        return;
    }
    ni = p->next[i];
    nj = p->next[j];
    T=TRUE;
    while (ni->Type !=J) {
        if (ni->Type != nj->Type){
            T=FALSE;
            break;
        }
        else if (ni->t>nj->t){
            T=FALSE;
            break;
        }
        ni = ni->next[0];
        nj = nj->next[0];
    }
    if (T) {
        delete p->next[i];
        for (k=i;k<(p->nb-1);k++)
            p->next[k] = p->next[k+1];
        j=j-1;
        p->nb = p->nb - 1;
```

```
        }

}
/*
unsigned PFG::apply_transformation(Node *n)
{
Node *p1,*p2;
unsigned l[2],c_l[2],s_l[2];
unsigned t[2],c_t[2],s_t[2];
unsigned select = 0;
unsigned i,j;

    i=0;j=1;
    while (i<((n->nb)-1)) {
        p1=n->next[i];p2=n->next[j];
        compute_lengths(p1,l[0],c_l[0],s_l[0]);
        compute_lengths(p2,l[1],c_l[1],s_l[1]);

        compute_times(p1,t[0],c_t[0],s_t[0]);
        compute_times(p2,t[1],c_t[1],s_t[1]);

        select = 0;
        if ((l[0]==0) || (l[1]==0))
            select=1;
        else if ((c_l[0]==0) && (c_l[1]==0))
            select = 2;
        else if ((c_l[0]==0) || (c_l[1]==0))
            select = 3;
        else if (l[0] == l[1]) select = 4;

        switch (select) {
        case 1: apply_T1(n,i,j);
            break;
        case 2: apply_T2(n,i,j);
            break;
        case 3: apply_T3(n,i,j,t[0],t[1],c_l[0]);
            break;
        case 4: apply_T4(n,i,j);
            break;
        default: error("Unimplemented transformation");
        }
        j++; if (j>=(n->nb)) { i++; j=i+1;}
    }
    if ((n->nb)==1) return TRUE; else return FALSE;
}
```

```
*/

unsigned min(unsigned a, unsigned b)
{if (a<b) return a; else return b;}

#define MAX_N_SEG      100

/*
    transforms two branches pointed by r and s if possible.
    n is the pointer to the corresponding fork node.

*/
unsigned PFG::apply_T(Node *n,unsigned &r, unsigned &s)
{
unsigned l[2],c_l[2],s_l[2];
unsigned t[2],c_t[2],s_t[2];
Node *p1,*p2;
unsigned i,p_l,q_l;
unsigned keep,throw;

struct {
    unsigned t;
    node_type T;
    unsigned id;
} p[MAX_N_SEG],q[MAX_N_SEG];
unsigned diff;

    p1=n->next[r];p2=n->next[s];
    compute_lengths(p1,l[0],c_l[0],s_l[0]);
    compute_lengths(p2,l[1],c_l[1],s_l[1]);

    compute_times(p1,t[0],c_t[0],s_t[0]);
    compute_times(p2,t[1],c_t[1],s_t[1]);
    if ((l[0]==0) || (l[1]==0)) {
        apply_T1(n,r,s);
        return TRUE;
    }
    if ((c_l[0]==0) && (c_l[1]==0)){
        apply_T6(n,r,s,t[0],t[1]);
        return TRUE;
    }
    if ((c_l[0]==0) || (c_l[1]==0)) {
        apply_T3(n,r,s,t[0],t[1],c_l[0]);
        return TRUE;
    }
```

```
if (l[0] > l[1] ){
    if (t[0] < t[1]) {
        return FALSE;
    } else {
        p_1 = l[0];q_1=l[1];
        for(i=0;p1->Type != J;i++) {
            p[i].t = p1->t;
            p[i].T = p1->Type;
            p[i].id = p1->id;
            p1 = p1->next[0];
        }
        for(i=0;p2->Type != J;i++) {
            q[i].t = p2->t;
            q[i].T = p2->Type;
            q[i].id = p2->id;
            p2 = p2->next[0];
        }
        throw=s;
        keep=r;
    }
} else if (t[1]<t[0]) {
        return FALSE;
    } else {
        p_1 = l[1];q_1=l[0];
        for(i=0;p1->Type != J;i++) {
            q[i].t = p1->t;
            q[i].T = p1->Type;
            q[i].id = p1->id;
            p1 = p1->next[0];
        }
        for(i=0;p2->Type != J;i++) {
            p[i].t = p2->t;
            p[i].T = p2->Type;
            p[i].id = p2->id;
            p2 = p2->next[0];
        }
        throw=r;
        keep=s;
    }

{unsigned trfm = TRUE;
for (i=0;i<q_1;i++){
    if ((p[i].t < q[i].t)  || (p[i].T!=q[i].T)) {
        trfm =FALSE; break;
    }
```

```
            if ((p[i].T==C) && (q[i].T==C) && (p[i].id != q[i].id)) {
                trfm = FALSE; break;
            }
    }
    if (trfm) {
            if (q_l!=0) delete n->next[throw];
            unsigned k;
            p1 = n->next[keep];
            for (k=throw;k<(n->nb-1);k++)
                n->next[k] = n->next[k+1];
            for (i=0;i<p_l;i++){
                p1->t = p[i].t;
                p1 = p1->next[0];
            }
            n->nb = n->nb - 1;
            s=s-1;
            return trfm;
    }}


    diff = p[0].t - q[0].t;
    for (i=1;i<q_l;i++) {
        if (p[i].T==C) {
            if ((diff>0) && ((p[i].t-q[i].t)<0)) {
                unsigned t;
                t = min(diff,abs(p[i].t - q[i].t));
                p[i].t   += t;
                p[i-1].t -=t;

                diff = p[i].t - q[i].t;
            } else diff = p[i].t - q[i].t;
        } else     {
            if ((diff<0) && (p[i].t > 0 )){
                unsigned t;
                t = min(p[i].t,abs(diff));
                p[i].t -= t;
                p[i-1].t += t;
            }
            diff = p[i].t - q[i].t;
        }
    }

    diff = q[0].t - p[0].t;
    for (i=1;i<q_l;i++) {
        if (q[i].T==C) {
            if ((diff>0) && ((q[i].t-p[i].t)<0)) {
```

```
                    unsigned t;
                    t = min(diff,abs(q[i].t - p[i].t));
                    cout<<"\n\n t"<<t<<"\n "<<flush;
                    q[i].t +=t;
                    q[i-1].t -=t;
                    diff = q[i].t - p[i].t;
                } else diff = q[i].t - p[i].t;
            } else    {
                if ((diff<0) && (q[i].t>0)) {
                    unsigned t;
                    t = min(q[i].t,abs(diff));
                    q[i].t -= t;
                    q[i-1].t += t;
                }
                diff = q[i].t - p[i].t;
            }
    }
    if (p_l>q_l) for (i=q_l;i<p_l;i++) q[q_l-1].t -= p[i].t;
    unsigned trfm = TRUE;
    for (i=0;i<q_l;i++){
        if ((p[i].t < q[i].t)  || (p[i].T!=q[i].T)) {
            trfm =FALSE; break;
        }
        if ((p[i].T==C) && (q[i].T==C) && (p[i].id != q[i].id)) {
            trfm = FALSE; break;
        }
    }
    if (trfm) {
        if (q_l!=0) delete n->next[throw];
        unsigned k;
        p1 = n->next[keep];
        for (k=throw;k<(n->nb-1);k++)
            n->next[k] = n->next[k+1];
        for (i=0;i<p_l;i++){
            p1->t = p[i].t;
            p1 = p1->next[0];
        }
        n->nb = n->nb - 1;
        s=s-1;
    }
return trfm;
}

/*
    applies the the transformations to all the branches (nb>1).
```

```
        return TRUE if all branches are transformed.
*/


unsigned PFG::apply_transformation(Node *n)
{
unsigned i,j;

    i=0;j=1;
    while (i<((n->nb)-1)) {
        apply_T(n,i,j);
        j++; if (j>=(n->nb)) { i++; j=i+1;}
    }
    if ((n->nb)==1) return TRUE; else return FALSE;
}


unsigned PFG::apply_minimize(Node *n)
{
unsigned i,j;
int a,b;

    i=0;j=1;
    while (i<((n->nb)-1)) {
        a=linear(n->next[i]);
        b=linear(n->next[j]);
        if (a && b){
            apply_T(n,i,j);
            j++; if (j>=(n->nb)) { i++; j=i+1;}
        } else {
            if (!a && !b) {i++;j=i+1;}
            else {
                if (!b) {j++;if (j>=(n->nb)) { i++;j=i+1;}}
                if (!a) {i++;j=i+1;}
            }
        }
    }
    if ((n->nb)==1) return TRUE; else return FALSE;
}


int PFG::linear(Node *n)
{
    while (n!=NULL){
        if (n->Type == J) return TRUE;
        if (n->Type == F) return FALSE;
        n=n->next[0];
```

```
    }
    error("\n Error : linear()");
    exit(0);
return FALSE;
}


void PFG::test(Node *n)
{
Node *prev;
    prev=NULL;
    while (n->Type !=F) {
        prev=n;
        n=n->next[0];
    }
    prev = adjust_branches(n,prev);
    *out<<"\n Sub graph  "<<flush;
    printf(prev);
    *out<<"\n End sub graph \n"<<flush;

    calc_depth(start,0);
}


/*
    recursively apply the transformations to a PFG pointed by n.
    returns TRUE if the transformations result in a liear sequence
    of nodes.
*/

unsigned PFG::transform(Node *n,Node *prev)
{
unsigned Trfm=TRUE;
    while (n!=NULL){
        switch (n->Type) {
        case S:
        case C:
            prev=n;
            n=n->next[0];
            break;
        case F:{
            unsigned Tr=TRUE;
            for (unsigned i=0;i<(n->nb);i++)
                Tr = transform(n->next[i],n) && Tr;
            if (Tr)  Tr = Tr && apply_transformation(n);
            if (!Tr) Tr = apply_minimize(n);
            if (Tr) {
```

```
                             prev = adjust_branches(n,prev);
                             n = prev->next[0];
                     } else {
                             prev = n->next[0];
                             while ((n->d) != (prev->d))
                                     prev=prev->next[0];
                             n = prev->next[0];
                     }
                     Trfm = Trfm && Tr;
                     }
                     break;
             case J:
                     return Trfm;
             case I:
                     error(" Invalid node ");
                     exit(0);
             }
     }
return Trfm;
}


double PFG::N_f()
{
    n_branches=0;
    n_branches_c=0;
    n_f(start);
return  ((double) n_branches);
}


void  PFG::n_f(Node *n)
{

    while (n!=NULL){
        switch (n->Type) {
        case S:
        case C:
            n=n->next[0];
            break;
        case F:
            {

            n_branches += n->nb;
            n_branches_c += 1;

            for (unsigned i=0;i<(n->nb);i++)
```

```
                    n_f(n->next[i]);

            Node *t;
            t=n->next[0];
            while ((n->d) != (t->d))
                t=t->next[0];
            n = t->next[0];
            }
            break;
        case J:
            return ;
        case I:
            error(" Invalid node ");
            exit(0);
        }
    }
return ;
}



double PFG::max_depth(Node *n)
{
double maxd=0;
    while (n!=NULL){
        switch (n->Type) {
        case S:
        case C:
            maxd = max(maxd,n->d);
            n=n->next[0];
            break;
        case F:
        {
            for (unsigned i=0;i<(n->nb);i++)
                maxd = max(max_depth(n->next[i]),maxd);
            Node *t;
            t=n->next[0];
            while ((n->d) != (t->d))
                t=t->next[0];
            n = t->next[0];
        }
            break;
        case J:
            return maxd;
        case I:
            error(" Invalid node ");
```

```
                exit(0);
            }
        }
return maxd;
}


double PFG::compute_paths(Node *n)
{
double paths=1;
    while (n!=NULL){
        switch (n->Type) {
        case S:
        case C:
            n=n->next[0];
            break;
        case F:
        {
            double pth=0;
            for (unsigned i=0;i<(n->nb);i++)
                pth += compute_paths(n->next[i]);
            Node *t;
            t=n->next[0];
            while ((n->d) != (t->d))
                t=t->next[0];
            n = t->next[0];
            paths = paths * pth;
        }
            break;
        case J:
            return paths;
        case I:
            error(" Invalid node ");
            exit(0);
        }
    }
return paths;
}


/*
    removes a Fork and Join node. Also simple segments are
    preclustered.
*/


Node *PFG::adjust_branches(Node *n,Node *prev)
{
```

```
Node *temp;
unsigned i;
    if ((n->Type != F) || (n->nb != 1)){
        error("Invalid adjustment of PFG");
        exit(0);
    }
    if (prev==NULL) start = n->next[0];
    else{ if (prev->Type == F) {
        i=0;
        while (prev->next[i]!=n) i++;
        prev->next[i]=n->next[0];
    }
    else prev->next[0]=n->next[0];
    }
    temp=n->next[0];(n->nb)=0;
    delete n;
    n = temp;
    if (prev!=NULL){
        if ((prev->Type ==S) && (n->Type == S)) {
            prev->t = prev->t + n->t;
            prev->next[0]=n->next[0];
            temp = n->next[0]; n->nb = 0;
            delete n;
            n = temp;
        }
    }
    while (n->Type != J) {
        prev = n;
        n = n->next[0];
    }
    if (prev==NULL) start = n->next[0];
    else {
        prev->next[0] = n->next[0];
        prev->nb = n->nb;
    }
    temp=n->next[0];(n->nb)=0;
    delete n;
    n = temp;
    if ((prev!=NULL) && (n!=NULL)){
        if ((prev->Type ==S) && (n->Type == S)) {
            prev->t = prev->t + n->t;
            prev->next[0]=n->next[0];
            prev->nb = n->nb;
            temp = n->next[0]; n->nb = 0;
            delete n;
```

```
                    n = temp;
            }
        }
return prev;
}


/*
    precluster a sequence of linear segments.
*/

Node *PFG::cluster_sequence(Node *n)
{
Node *prev;
    if (n==NULL) return n;
    while ((n->Type !=J) && (n->Type !=F)) {
        prev = n;
        n = n->next[0];
        if (n==NULL) return n;
        if ((prev->Type == S) && (n->Type == S)) {
            prev->t = prev->t + n->t;
            prev->next[0]=n->next[0];
            prev->nb = n->nb;
            n->nb = 0; n->next[0] = NULL;
            delete n;
            n = prev;
        }
    }
return n;
}


/*
    precluster a PFG pointed by n.
*/
void PFG::cluster_PFG(Node *n)
{
    while (n != NULL) {
        switch (n->Type) {
        case S:
        case C:
            n = cluster_sequence(n);
            break;
        case F:
            for (unsigned i=0;i<(n->nb); i++)
                cluster_PFG(n->next[i]);
            unsigned d;
```

```
            d = n->d; n = n->next[0];
            while ((n->d) != d)
                n = n->next[0];
            if (n->Type != J) {
                error("Error: J node expected");
                exit(0);
            }
            n = n->next[0];
            break;
        case J:
            return;
        case I:
            error("Error : Invalid type ");
            exit(0);
        }
    }
}

/*

    The next set of functions are used to read a PFG from a
    file.
*/

char PFG::skip()
{
char ch;
    *in>>ch;
#ifdef PRINT
    lc++;if (lc>70) {cout<<"\n";lc=0;} cout<<ch<<flush;
#endif
    while ((ch != '[') && (ch  != ']') && (ch != '}') &&
        (ch!='{') && (ch!='(') && (ch!=')') && (!(in)->eof())){
        *in>>ch;
#ifdef PRINT
        lc++;if (lc>70) {cout<<"\n";lc=0;} cout<<ch<<flush;
#endif
    }
return ch;
}

node_type PFG::get_type()
{
char ch,ch1;
    *in>>ch;
```

```
    *in>>ch1;
#ifdef PRINT
    lc++;if (lc>70) {cout<<"\n";lc=0;}  cout<<ch;
    lc++;if (lc>70) {cout<<"\n";lc=0;} cout<<ch1<<flush;
#endif
    switch (ch) {
    case 'F': return F;
    case 'J': return J;
    case 'S': return S;
    case 'C': return C;
    default : error("Invalid Node type ");
            exit(0);
    }
}


unsigned PFG::get_time()
{
unsigned i;
    *in>>i;
#ifdef PRINT
    lc++; if (lc>70) {cout<<"\n";lc=0;} cout<<i;
#endif
return i;
}
void PFG::get_id(unsigned &id)
{
char ch;
    *in>>ch;
    if (ch != ',') {
        error("\n error : id expected");
        exit(0);
    }
    *in>>id;
    if (id == 0) {
        error("\n error : id should be non zero");
        exit(0);
    }
#ifdef PRINTF
    lc++; if (lc>70) {cout<<"\n";lc=0;} cout<<ch;
    lc++; if (lc>70) {cout<<"\n";lc=0;} cout<<id;
#endif
}

void PFG::exit_eof(char *s)
{
```

```
    if ((in)->eof()) {
        error("\npremature eof:",s);
        exit(0);
    }
}


void PFG::node_S_C(node_type nt,unsigned t,Node **new_node,
        Node **n,char &ch)
{
unsigned id=0;
    if (nt == C) get_id(id);
    *new_node = new Node(nt,t,id);

    add_node(*new_node,*n);
    *n=*new_node;
    ch = skip();ch = skip();
}


void PFG::node_F(node_type nt,unsigned t,Node **new_node,
        Node **n,char &ch)
{
    *new_node = new Node(nt,t);
    add_node(*new_node,*n);
    *n=*new_node;
    skip();exit_eof();ch = skip();
    exit_eof();
    if (ch!='{') {
        error("Error: { expected");
        exit(0);
    }
    Node *t1 = new Node(J,0);
    ch=skip();
    while (ch!='}'){
        exit_eof();
        if (ch != '[') {
            error("error: [ - expected ");
            exit(0);
        }
        *new_node=read_PFG(*n);
        add_node(t1,*new_node);
        ch = skip();
    }
    *new_node=t1;*n=t1;ch=skip();
}
```

```
Node *PFG :: read_PFG(Node *n)
{
char ch;
node_type nt;
Node *new_node;
unsigned t;
    ch =skip();
    if ((in)->eof()) return n;
    if (ch == ']') return n;
    while (ch == '(') {
        nt =get_type();
        t = get_time();
        switch (nt) {
        case S:
        case C:
            node_S_C(nt,t,&new_node,&n,ch);
            break;
        case F:
            node_F(nt,t,&new_node,&n,ch);
            break;
        case J:
            ch=skip();ch=skip();
            break;
        case I:
            error("\nInvalid type in function read_PFG()");
            exit(0);
        }
    }
return n;
}


PFG :: ~PFG()
{
    if (start != NULL ) delete start;
    out = NULL;
    if (in!=NULL){
        (in)->close();
        delete in;
    }

}

PFG :: PFG()
{
    start=NULL;
```

```
    in=NULL;
    out=NULL;
}

PFG :: PFG(ofstream *fout)
{
    start = NULL;
    /* out = new ofstream(fout); */
    in = NULL;
    out = fout;
    if (!*out) {
        error("cannot open output file");
        exit(0);
    }
}
PFG :: PFG(char *fin,ofstream *fout)
{
    start = NULL;
    in = new ifstream(fin);
    if (!*in) {
        error("cannot open input file");
        exit(0);
    }
    /*out = new ofstream(fout); */
    out = fout;
    if (!*out) {
        error("cannot open output file");
        exit(0);
    }
    strcpy(infile,fin);
}

#define RAND_RES 100

/*
    Generates a random PFG with the appropriate properties.
*/

Node * PFG::generate(unsigned M[],unsigned l,unsigned bf,float pe,
        float fp,float cp,float sp,Node *n,unsigned d,unsigned p_o)
{
Node *new_node,*t1,*t2,*t3;
unsigned rnd,rnd1,i,j;
unsigned RandomChoice[RAND_RES+1];
unsigned RandomChoice2[RAND_RES+1];
```

```
unsigned Max;
unsigned randchild;
unsigned c,s,f;

    Max=M[d];
    if (d>1) {
        cp=fp/2;
        sp=fp/2;
        fp=0;
    }

    for (i=0;i<RAND_RES*pe;i++) RandomChoice2[i]=0;
    for (i=(int)(RAND_RES*pe);i<RAND_RES;i++) RandomChoice2[i]=1;
    for (i=0;i<RAND_RES*fp;i++) RandomChoice[i]=1;
    for (i=(int)(RAND_RES*fp);i<RAND_RES*fp+RAND_RES*cp;i++)
        RandomChoice[i]=2;
    for (i=(int)(RAND_RES*fp+RAND_RES*cp+0.5);i<RAND_RES;i++)
        RandomChoice[i]=3;
    unsigned flag =0;
    c=s=f=0;
    for (i=0;(c+s+f)<Max;i++) {
        rnd=rand()%RAND_RES;
        while ((RandomChoice[rnd]==1) && (d>=1)) rnd=rand()%RAND_RES;
        if (d==0) rnd=1;
        switch (RandomChoice[rnd]) {
        case 2: if (p_o) cout<<"\n(C,1) ";
            new_node=NULL;
            new_node = new Node(C,1,1);
            if (new_node==NULL){error("C");exit(0);}
            add_node(new_node,n);
            n=new_node;
            c++;
            flag=1;
            break;
        case 1:
            if ((d<MAX_DEPTH) && (d<=1))
            {
                t1=NULL;
                t2=NULL;
                t2=new Node(J,0);
                if (t2==NULL){error("t2"); exit(0);}
                t3=NULL;
                t3 =new Node(F,0);
                if (t3==NULL){error("t3");exit(0);}
                add_node(t3,n);
```

```
                    n=t3;
                    if(p_o) cout<<"\n(F,0){"<<flush;
                    randchild = 2 + ( rand()%(bf-1) );
                    int Empty = FALSE;
                    for (j=0;j<randchild;j++){
                        rnd1=rand()%RAND_RES;
                        if (RandomChoice2[rnd1]==1 || Empty){
                        if (p_o)cout<<"\n["<<flush;
                            t1=generate(M,1,
                            bf,pe,fp,cp,sp,n,(d+1));
                        if (p_o) cout<<"\n]"<<flush;
                        }
                        else {
                            t1=n;
                             Empty =TRUE;
                        }
                        add_node(t2,t1);
                    }
                    if (p_o) cout<<"\n}";
                    if (p_o) cout<<"\n(J,0)"<<flush;
                    n=t2;
                    flag =1;
                    f++;
                 }
              flag=1;
              break;
           case 3: if (p_o) cout<<"\n(S,1)";
               new_node=NULL;
               new_node = new Node(S,1);
               if (new_node==NULL) {error("Error: Null ptr ");exit(0);}
               add_node(new_node,n);
               n=new_node;
               s++;
               flag=1;
               break;
           default: error("Invalid choice");
               exit(0);
           }
           if (d==0) break;
       }
   if (!flag) {
       cout<<"\n\n\nInvalid parameters: too many Nuls"<<flush;
       exit(0);
   }
return n;
```

```
}

void PFG::calc_depth(Node *n,unsigned d)
{
unsigned i;
    if (d>100) {
        error("Somethings wrong");
        exit(0);
    }
    while (n!=NULL) {
        while (((n)->Type !=F) && ((n)->Type!=J)){
            (n)->d=d;
            n=n->get_next(0);
            if (n==NULL) return;
        }
        if ((n)->Type==F){
            (n)->d=d;
            for (i=0;i<(n)->nb;i++){
                calc_depth((n)->get_next(i),d+1);
            }
            n=(n)->get_next(0);
            while ((n)->d != d) n=(n)->get_next(0);

            if ((n)->Type != J){
                error("Error in graph J expected");
                exit(0);
            }
            n=(n)->get_next(0);

        } else
        if ((n)->Type==J){
            (n)->d=d-1;
            return;
        } else {
            error("Cannot be ");
            exit(0);
        }
    }
}


void PFG::add_node(Node *n,Node *at)
{
    if (at==NULL) {
        start = n;
```

```
        (n)->rc=1;
    } else (at)->add_nd(n);
}


void print_tab(unsigned gps,ofstream *out)
{
unsigned i;
    for (i=0;i<gps;i++) str[i]=32;
    str[gps]=0;
    *out<<"\n"<<str<<flush;
}


void print_enum(node_type nt,ofstream *out)
{
    switch (nt) {
    case F: *out<<" F ";
        break;
    case J: *out<<" J ";
        break;
    case C: *out<<" C ";
        break;
    case S: *out<<" S ";
        break;
    case I: cout<<"\n\n Error at print_enum ";
            exit(0);
    }
    *out<<flush;
}


void print_tuple(Node *n,ofstream *out)
{
    *out<<"("<<flush;
    print_enum((n)->Type,out);
    *out<<","<<(n)->t;
    if ((n->Type==C)) *out<<","<<n->id<<","<<n->rt<<","<<n->delay;
    *out<<")"<<flush;
}


void PFG::printf(Node *n)
{
unsigned i;
Node *n1;

    if (n!=NULL){
```

```
while (((n)->Type !=F) && ((n)->Type !=J)){
    print_tab(gps,out);
    print_tuple(n,out);
    n=(n)->get_next(0);
    if (n==NULL) break;
}

if (n!=NULL){
if ((n)->Type == F) {
    gps +=4;
    print_tab(gps-4,out);
    print_tuple(n,out);
    *out<<" {"<<flush;
    for (i=0;i<((n)->nb);i++){
        print_tab(gps,out);
        *out<<"["<<flush;
        gps +=4;
        printf((n)->get_next(i));
        gps -=4;
        print_tab(gps,out);
        *out<<"]"<<flush;
    }
    gps -= 4;
    print_tab(gps,out);
    *out<<"}"<<flush;
    if (n==NULL) {
        error(" nullpointer");
        exit(0);
    }
    n1 = (n)->get_next(0);
    if (n1==NULL) {
        error(" null pointer");
        exit(0);
    }
    while (
        ((n1)->d != (n)->d) ){
        n1=n1->get_next(0);
    }
    print_tab(gps,out);
    *out<<flush;
    print_tuple(n1,out);
    printf((n1)->get_next(0));
    return;
}
if ((n)->Type == J) {
```

```
            return;
        }}
    }
}
```

```
/*
    The class node represents a node in the PFG with the appropriate
    fields.
*/


#define     MAX_N_COND      8

/* Critical, Non-Critical,Fork,Join */
enum node_type  {C,S,F,J,I};

class Node {
    public:
    node_type Type;
    unsigned id;     /* resource id */
    unsigned t;      /* Units of time */
    unsigned rt;     /* release time */
    unsigned nb;     /* Number of Branches */
    unsigned d;      /* Depth in the tree */
    unsigned delay;
    unsigned rc;     /* reference counter */
    Node *next[MAX_N_COND];
    Node();
    Node(node_type T);
    Node(node_type T,unsigned t1);
    Node(node_type T,unsigned t1,unsigned i_d);
    ~Node();
    void print();
    void add_nd(Node *n);
    Node *get_next(unsigned i);
};

Node* Node:: get_next(unsigned i)
{ return next[i];}

Node ::~Node()
{
    for (unsigned i =0;i<nb;i++){
        ((next[i])-> rc )--;
        if ((next[i])->rc==0) delete next[i];
    }
}
Node :: Node()
{
```

```
    Type = I;rc=t=d=nb=rt=id=delay=0;
    for (unsigned i=0;i<MAX_N_COND;i++) next[i]=NULL;
}


Node ::    Node(node_type T)
{
    Type = T;t=nb=d=rc=rt=id=delay=0;
    for (unsigned i=0;i<MAX_N_COND;i++) next[i]=NULL;
}


Node ::    Node(node_type T,unsigned t1)
{
    Type = T;t=t1;nb=d=rc=rt=id=delay=0;
    for (unsigned i=0;i<MAX_N_COND;i++) next[i]=NULL;
}


Node ::    Node(node_type T,unsigned t1,unsigned i_d)
{
    Type = T;t=t1;nb=d=rc=rt=id=delay=0;
    id = i_d;
    for (unsigned i=0;i<MAX_N_COND;i++) next[i]=NULL;
}


void Node::add_nd(Node *n)
{
    next[nb] = n;
    nb++;
    (n)->rc=((n)->rc)+1;
    if (nb>=(MAX_N_COND-1)) {
        error("MAX N COND exceeded");
        exit(0);
    }
}


void Node::print()
{
    cout << "\n Type: ";
    switch (Type) {
        case I: cout<<"Invalid node        ";
            break;
        case C: cout<<"Critical section    ";
            break;
        case S: cout<<"Non Critical section ";
            break;
        case J: cout<<"Join                ";
```

```
            break;
        case F: cout<<"Fork                ";
            break;
    }
    cout<<", d: "<<d<<", t: "<<t<<", nb: "<<nb<<", rc: "<< rc<<flush;
    cout<<", rt "<<rt<<", id "<<id<<flush;
}
```

```
class PFG;
class TreeNode;
#define MAX_BRANCHES      128
#define MAX_PFGS          10

#include "list.h"

#define MAX_N_RESOURCE    200
class Traversal {
    public:
    static unsigned g_count;
    unsigned time_static;
    unsigned time;
    PFG *G;
    TreeNode *root;
    TreeNode *row[MAX_BRANCHES];
    unsigned nl;               /* Number of leaves */

    Traversal(char *input,ofstream *fout);
    ~Traversal();
    void DFS(TreeNode *t);
    void print();
     void advance(unsigned r,unsigned *);

};

class PFGS { /* Program Flow Graphs */
    public:
    List L;
    unsigned lrt[MAX_N_RESOURCE];
    Traversal *T[MAX_PFGS];
    unsigned np,x;
    PFGS(unsigned n, char *arg[],ofstream *fout);
    ~PFGS();
    TreeNode *row(unsigned i,unsigned j);/* ith graph, jth column */
    Node *node(unsigned i,unsigned j);    /* ith graph, jth column node*/
    void advance(unsigned i,unsigned j);
    void release_time_static(unsigned x);
    void release_time(unsigned x);
    void compute_release_times(unsigned minj,unsigned mini);
    void extract_items(unsigned minj,unsigned mini);
    void print();

};
```

```
void PFGS::extract_items(unsigned minj,unsigned mini)
{
unsigned i,j,id;
list_item li;

    id = node(minj,mini)->id;
    for (j=0;j<np;j++){
        for (i=0;i<T[j]->nl;i++){
            if (((mini!=i)||(minj!=j)) &&(id==node(j,i)->id)){
            li.i = i;li.j=j;
            li.t=row(j,i)->t;
            li.k=node(j,i)->t;
            li.r=li.t+li.k;
            L.insert(li);
                }
            }
        }
}
/*
    computes the release times of a set of resources.
    it is assumed that a request may get delayed up to
    x units of time.
*/

void PFGS::compute_release_times(unsigned minj, unsigned mini)
{
unsigned a,b,c,start;
unsigned i;
unsigned id;
        a=row(minj,mini)->t;
        start = a;
        b = node(minj,mini)->t;
        c = a+b;
        for (i=0;i<L.n;i++){
                if (( L.L[i].t) <= c){
                    row(L.L[i].j,L.L[i].i)->flag=TRUE;
                    c = c+L.L[i].k;
                }
        }
        for (i=0;i<L.n;i++)
            if (row(L.L[i].j,L.L[i].i)->flag) node(L.L[i].j,L.L[i].i)->rt=c+x;

        id = node(minj,mini)->id;
        node(minj,mini)->rt = c+x;
```

```
            if (lrt[id]<(c+x))
                lrt[id]=c+x;
            row(minj,mini)->flag=TRUE;
}


/*
void PFGS::compute_release_times(unsigned minj, unsigned mini)
{
unsigned a,b,c,start;
unsigned k,i;
unsigned id;
            a=row(minj,mini)->t;
            start = a;
            b = node(minj,mini)->t;
            c = a+b;
            for (i=0;i<L.n;i++){
                if (L.extract(i).j !=minj){
                    if (( L.L[i].t) <= c)
                        c = c+L.L[i].k;
                }
            }
            id = node(minj,mini)->id;
            node(minj,mini)->rt = c+x;
            if (lrt[id]<(c+x))
                lrt[id]=c+x;
            row(minj,mini)->flag=TRUE;
            int ti,tj;
            for (k=0;k<L.n;k++){
                tj=L.L[k].j;
                ti=L.L[k].i;
                L.L[k].i=mini;
                L.L[k].j=minj;
                mini=ti;minj=tj;
                L.L[k].t = a;
                L.L[k].k = b;
                L.L[k].r = c;
                a=row(minj,mini)->t;
                b = node(minj,mini)->t;
                // c = a+b;
                c = start+b;
                for (i=0;i<L.n;i++){
                    if (L.L[i].j !=minj){
                    if ( ((L.L[i].t>=start) &&(L.L[i].t <= c)) ||
                            ((L.L[i].r>=start) &&(L.L[i].r<= c)) ||
                        ((start>=L.L[i].t) &&(c<=L.L[i].r)) )
```

```
                                    c = c+L.L[i].k;
                        }
                }
                if (c > (a+b)) {
                        row(minj,mini)->flag=TRUE;
                        node(minj,mini)->rt = c+x;
                        if (lrt[id]<(c+x))
                                lrt[id]=c+x;
                }
        }
}
*/


void PFGS::release_time(unsigned comm_delay)
{
int f = FALSE;
int i,j;
int mini,minj;
unsigned mint;
unsigned tmp;

    x=comm_delay;
    for (i=0;i<MAX_N_RESOURCE;i++) lrt[i]=0;
    for (j=0;j<np;j++){
        if (T[j]->root!=NULL)
            delete T[j]->root;
        T[j]->row[0] = new TreeNode(NULL,T[j]->G->start,0);
        T[j]->root =T[j]->row[0];
        T[j]->nl=1;
        if (node(j,0)->Type != C) advance(j,0);
    }

    int ptr=np;
    while (!f) {
        /* Finds the earliest resource request from the list of
           resource requests.
        */
        mini=0;minj=0;mint=row(0,0)->t;
        for (j=0;j<np;j++){
            for (i=0;i<T[j]->nl;i++){
                if ((row(j,i)->t) < (lrt[node(j,i)->id])) {
                        node(j,i)->delay=lrt[node(j,i)->id]-row(j,i)->t;
    row(j,i)->t=lrt[node(j,i)->id];
                }
                if (mint>(row(j,i)->t)) {
```

```
                                mini=i;minj=j;
                                mint=row(j,i)->t;
                        }
                }
        }
        L.n=0;
        extract_items(minj,mini);
        compute_release_times(minj,mini);
        for (j=0;j<np;j++){
            tmp = T[j]->nl;
            for (i=0;i<min(tmp,T[j]->nl);i++){
                if (row(j,i)->flag){
                    row(j,i)->flag=FALSE;
                    advance(j,i);
                    i--;
                }
            }
        }
        /*
            If the program terminates as a result of advancing a resource
            request then that PFG is not included for the calculation of the
            release times. If processes are periodic then after idling till
            processes framtime it must be re-include in the calculation of
            release times.
        */
        for (j=0;j<np;j++)
            if (node(j,0)==NULL) {
                T[j]->time=row(j,0)->t;
                Traversal *Tmp = T[j];
                T[j] = T[np-1];
                T[np-1]=Tmp;
                np--;
                j--;
            }
        if (np==0) f =TRUE;
    }
    np=ptr;
}


/*
    computes the release times without any contention for the
    processes. This is used to compute the slow down.
*/

void PFGS::release_time_static(unsigned comm_delay)
```

```
{
int f=FALSE;
int tmp,i,j;
int ptr;
    for (i=0;i<MAX_N_RESOURCE;i++) lrt[i]=0;
    for (j=0;j<np;j++){
        if (T[j]->root!=NULL)
            delete T[j]->root;
        T[j]->row[0] = new TreeNode(NULL,T[j]->G->start,0);
        T[j]->root =T[j]->row[0];
        T[j]->nl=1;
        if (node(j,0)->Type != C) advance(j,0);
    }
    ptr = np;
    while (!f) {
        for (j=0;j<np;j++){
            for (i=0;i<T[j]->nl;i++){
                node(j,i)->rt=row(j,i)->t+node(j,i)->t+comm_delay;
            }
        }
        for (j=0;j<np;j++){
            for (i=0;i<T[j]->nl;i++)
                row(j,i)->flag=TRUE;
        }
        for (j=0;j<np;j++){
             tmp = T[j]->nl;
            for (i=0;i<min(tmp,T[j]->nl);i++){
                if (row(j,i)->flag){
                    advance(j,i);
                    i--;
                }
            }
        }
        for (j=0;j<np;j++)
            if (node(j,0)==NULL) {
                T[j]->time_static=row(j,0)->t;
                Traversal *Tmp = T[j];
                T[j] = T[np-1];
                T[np-1]=Tmp;
                np--;
                j--;
            }
        if (np==0) f =TRUE;

    }
```

```
        np =ptr;
}

/*
    Advances the j th resource request in the i th PFG.
    Last resource release  time is used to compute the delay
    prior to making a request.
*/

void PFGS::advance(unsigned i,unsigned j)
{
    T[i]->advance(j,lrt);
}
PFGS::~PFGS()
{
     for (int i=0;i<np;i++)
         delete T[i];
}

PFGS::PFGS(unsigned n,char *arg[],ofstream *fout)
{
    np = n;
    for (int i=0;i<np;i++)
        T[i]=new Traversal(arg[i],fout);
}

/*
    Returns a pointer to the node of the i th PFG corresponding
    to the j th resource request.
*/

Node *PFGS::node(unsigned i,unsigned j)
{
    return ((T[i])->row[j])->node;
}

/*
    Returns the pointer the leaf node in the TOP (tree of Pointers)
    of the i th tree correponding to the i th PFG and the jth
    resource resource request
*/

TreeNode *PFGS::row(unsigned i,unsigned j)
{
    return ((T[i])->row[j]);
```

```
}

void PFGS::print()
{
    for (int i=0;i<np;i++)
        T[i]->print();
}

void Traversal::print()
{
    *(G->out)<<"\n---------------------------------------------------------\n";
    G->print("\n(Type,t,{id,rt,delay})");
    *(G->out)<<"\nPFG::"<<G->infile<<"\n";
    G->printf(G->start);
    *(G->out)<<"\n";
    *(G->out)<<"\nTermination time      = "<<time;
    *(G->out)<<"\nSlow Down             = "<<((float)time/time_static);
}

/*
    This advances the r th resource request to the next request in the
    same execution path as request r. Last release time lrt[j] is used
    to compute any delay if necessary.
*/

void Traversal::advance(unsigned r,unsigned *lrt)
{
Node *n;
TreeNode *tn;
unsigned i;
int found =FALSE;
    row[r]->flag=FALSE;
    n = (row[r])->node;
    if (n==NULL) {return;}
    if (n->Type==C) {
        (row[r])->t = n->rt;
        n = n->next[0];
        (row[r])->node = n;
    }
    if (n==NULL) {return;}
    tn = row[r];
    for (i=r;i<(nl-1);i++)
        row[i]=row[i+1];
    row[nl-1]=tn;
    r =nl-1;
```

```
while (!found) {
    switch (n->Type) {
    case S:
        (row[r])->t += n->t;
        n = n->next[0];
        (row[r])->node = n;
        break;
    case C:
        if (((row[r])->t) < (lrt[n->id])){
            n->delay = lrt[n->id]-(row[r])->t;
            (row[r])->t = lrt[n->id];
        }
        n->rt = 0;
        (row[r])->node = n;
        r++;
        if (r<nl) n=(row[r])->node;
        break;
    case F:
        for (i=0;i<n->nb;i++){
            tn = new TreeNode(row[r],n->next[i],(row[r])->t);
            (row[r])->add_node(tn);
            row[nl] = tn;
            nl++;
            }
        for (i=r;i<(nl-1);i++)
            row[i]=row[i+1];
        nl--;

        n = (row[r])->node;
        break;
    case J:
        nl--;
        tn = (row[r])->parent;
        delete row[r];
        {
        for (int j = r;j<nl;j++)
            row[j]=row[j+1];
        }
        if ((tn->nc)==0){
            n->t = tn->t;
            n = n->next[0];
            row[nl]=tn; nl++;
            tn->node = n;
        }
        if (r<nl) {
```

```
                    n = (row[r])->node;
            }
            break;
        case I:
            error("\n Error:advance(): Invalid node ");
            exit(0);
        }
        if (r>=nl) found=TRUE;
        if (n==NULL) found=TRUE;
    }
}


void Traversal::DFS(TreeNode *t)
{
    if (t==NULL) return;
    for (int i=0;i<t->nc;i++)
        DFS(t->child[i]);
    t->node->print();
}

Traversal::Traversal(char *input,ofstream *fout)
{
    g_count++;
    G = new PFG(input,fout);
    G->read_PFG(NULL);
    G->calc_depth(G->start,0);
    G->cluster_PFG(G->start);
    G->transform(G->start);
    G->calc_depth(G->start,0);
    if (G->start!=NULL) {
        row[0] = new TreeNode(NULL,G->start,0);
        root =row[0];
        nl=1;
    } else nl = 0;
}

Traversal::~Traversal()
{
    delete G;
    delete row[0];
    g_count--;
}
```

```
class Node;

#define MAX_BF      128

class TreeNode {
    public:
    unsigned flag;          /* Boolean */
    TreeNode *parent;       /* pointer to the parent Node */
    unsigned nc;            /* number of children */
    unsigned t;             /* time at this fork Node in the PFG */
    TreeNode *child[MAX_BF];/* Maximum branching factor */
    Node *node;             /* ptr to a node in a PFG */

    TreeNode(TreeNode *p,Node *n,unsigned time);
    ~TreeNode();
    void add_node(TreeNode *node);
};

/*
    Add a node (child ) to " this node ".
*/
void TreeNode::add_node(TreeNode *node)
{
    node->parent = this;
    child[nc]=node;
    nc++;
}

TreeNode::TreeNode(TreeNode *p,Node *n,unsigned time)
{
    for (nc=0;nc<MAX_BF;nc++)
        child[nc]=NULL;
    nc=0;
    parent = p;
    t = time;
    node=n;
    flag = FALSE;
}
TreeNode::~TreeNode()
{
    if (nc != 0) {
        error("\n Error at ~TreeNode(): TreeNode still intact\n");
        exit(0);
    }
```

```
        if (parent != NULL) {
            unsigned i;
            for (i=0;i<(parent->nc);i++)
                if ((parent->child[i])==this) break;
            if (this!=(parent->child[i])) {
                error("\nError at ~TreeNode():No such child pointer\n");
                exit(0);
            }
            unsigned j;
            for (j=i;j<((parent->nc)-1);j++)
                parent->child[j]=parent->child[j+1];
            (parent->nc)--;
            if (parent->t < t) parent->t = t;
        }
}
```

```
/*
    Orderd list of requests.
    The list is ordered by the field t, i.e. request time
*/


typedef struct list_item {
    unsigned i,j;  /* ith PFG jth pointer to request. */
    unsigned t;    /* request time */
    unsigned k;    /* resource usage time */
    unsigned r;    /* release time for this request */
};


class List {
    public:
    list_item L[MAX_BRANCHES*MAX_PFGS];
    unsigned n;     /* number of items in list */

    List() { n=0;}
    void insert(list_item item);
    list_item extract(unsigned i);
    void reset(void) { n=0; }
};


void List::insert(list_item item)
{
int i,j;
    if (n==0) {
        L[n]=item;
        n++;
        return;
    }
    for (i=0;(i<n) && (L[i].t < item.t);i++);
    if (i>n) L[++n]=item;
    else {
        for (j=n-1;j>=i;j--) L[j+1]=L[j];
        L[i]=item;
        n++;
    }
}


list_item List::extract(unsigned i)
{
    if (i>=n){
        cout<<"\n Error: List:extract() - invalid range \n"<<flush;
```

```
            exit(0);
        }
return L[i];
}
```

```
/*
    Generates  random PFGs in the space defined by s,1-fp+s,
    for 0<= s <= 1-fp and applies the transformation and computes the
    average efficiency.  Each point has a sample size of MAX_SAMPLES.
    s is sub divied in to MAX_DIV.
*/

#include <iostream.h>
#include <fstream.h>
#include <libc.h>

#define TRUE      1
#define FALSE     0

extern "C" {
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
}
void error(char *s,char *s2="");
char *arg[100];

#include "graph.h"

ofstream *initialise(char *[]);
void random_generate();

int MAX_DIVISIONS,MAX_SAMPLES;
double fpg;

main(int argc, char *argv[])
{
    fpg=0.33;MAX_DIVISIONS=10;MAX_SAMPLES=10;
    switch (argc) {
     case 4: MAX_SAMPLES=atoi(argv[3]);
     case 3: MAX_DIVISIONS=atoi(argv[2]);
     case 2: fpg = atof(argv[1]);
    }

    random_generate();
}
```

```
void random_generate()
{
unsigned M[10];
double e,p1,p2;
unsigned i,j;
Node *n;
ofstream *fout;
double del,e1;
double fp,s;
unsigned d,bf;

    e1=0; del=0;
    fp=fpg;d=5;bf=3;s=0;
    del = (1-fp)/MAX_DIVISIONS;
    s = -del;
    cout<<"\n\n Max Divisions = "<<MAX_DIVISIONS;
    cout<<"\n Max Samples  = "<<MAX_SAMPLES<<flush;
    cout<<"\nd="<<d<<" bf="<<bf<<" fp="<<fp<<" del="<<del<<"\n"<<flush;
    M[0]=1;M[1]=8;M[2]=6;M[3]=4;M[4]=4;M[5]=4;M[6]=4;
    fout = new ofstream("Output");
    for (i=0;i<=MAX_DIVISIONS;i++) {
        s += del;
        e=0;
        for (j=0;j<MAX_SAMPLES;j++){
            PFG G(fout);
            srand(rand() % 1000 + 1);
            n=G.generate(M,d,bf,0,fp,s,1-fp-s,NULL,0);
            G.calc_depth(G.get_start(),0);
            G.cluster_PFG(G.get_start());
            p1 = G.N_f();
            G.transform(G.get_start());
            p2 = G.N_f();
            e = e + (p1-p2)/(p1);
        }
        e1 +=e/MAX_SAMPLES;
        cout << "\n "<< s <<flush;
        cout<<"  "<<e/MAX_SAMPLES<<flush;
    }

        cout << "\n\nNumber of samples = "<<MAX_DIVISIONS<<"\n";
        cout << "Efficiency = " << (double) e1/MAX_DIVISIONS<<"\n"<<flush;
        delete fout;
}

ofstream *initialise(char *argv[])
```

```
{
ofstream *fout;
    fout = new ofstream(argv[1]);
    return fout;
}

void error(char *s,char *s2 )
{
    cerr << s << " " << s2 << "\n "<<flush;
}
```

```c
#include <iostream.h>
#include <fstream.h>
#include <libc.h>

#define TRUE        1
#define FALSE       0

extern "C" {
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
}
double get_min_E(double fp,unsigned d,unsigned bf);
double get_min_E1(double fp,unsigned d,unsigned bf);
void error(char *s,char *s2="");
char *arg[100];

#include "graph.h"

ofstream *initialise(char *[]);
void random_generate();

main(int argc, char *argv[])
{

    random_generate();
}

#define MAX_DIVISIONS  10
#define MAX_SAMPLES    100

void random_generate()
{
double fp;
unsigned d,bf;

    fp=0.1;d=5;bf=3;
    for (fp=0.1;fp<=0.5;fp += 0.04){
        cout << "\n "<<fp<<" "<<get_min_E1(fp,d,bf)<<flush;
    }
    cout<<"\n\n"<<flush;
}

double get_min_E(double fp,unsigned d,unsigned bf)
```

```
{
unsigned M[10];
double e,p1,p2;
unsigned i,j;
Node *n;
double del;
double s;
double E=1;
int FLAG=FALSE;

    del=s=0;
    del = (1-fp)/MAX_DIVISIONS;
    M[0]=1;M[1]=12;M[2]=8;M[3]=4;M[4]=4;M[5]=4;M[6]=4;
    for (i=0;(i<MAX_DIVISIONS-1) && !FLAG;i++) {
        FLAG=TRUE;
        s += del; e=0;
        for (j=0;j<MAX_SAMPLES;j++){
            PFG G;
            srand(rand() % 1000 + 1);
            n=G.generate(M,d,bf,0,fp,s,1-fp-s,NULL,0);
            G.calc_depth(G.get_start(),0);
            G.cluster_PFG(G.get_start());
            p1 = G.N_f();
            G.transform(G.get_start());
            p2 = G.N_f();
            e = e + (p1-p2)/(p1);
        }
        e = e/MAX_SAMPLES;
        if (E>e) {
            E = e;
            FLAG=FALSE;
        }
    }
return E;
}


double get_min_E1(double fp,unsigned d,unsigned bf)
{
unsigned M[10];
double e,p1,p2;
unsigned j;
double E=1;
    e=0;
    M[0]=1;M[1]=8;M[2]=6;M[3]=4;M[4]=4;M[5]=4;M[6]=4;
        for (j=0;j<MAX_SAMPLES;j++){
```

```
            PFG G;
            srand(rand() % 1000 + 1);
            G.generate(M,d,bf,0,fp,(1-fp)/2,(1-fp)/2,NULL,0);
            G.calc_depth(G.get_start(),0);
            G.cluster_PFG(G.get_start());
            p1 = G.N_f();
            G.transform(G.get_start());
            p2 = G.N_f();
            e = e + (p1-p2)/(p1);
        }
        E = e/MAX_SAMPLES;
return E;
}


ofstream *initialise(char *argv[])
{
ofstream *fout;
    fout = new ofstream(argv[1]);
    return fout;
}


void error(char *s,char *s2 )
{
    cerr << s << " " << s2 << "\n "<<flush;
}
```

```c
/*
    Generates the efficiency for the complete space defined by
    fp,cp,sp. MAX_SAMPLES samples are generated for each point.
*/
#include <iostream.h>
#include <fstream.h>
#include <libc.h>

#define TRUE    1
#define FALSE       0
extern "C" {
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
}
double get_min_E(double fp,unsigned d,unsigned bf);
double get_min_E1(double fp,unsigned d,unsigned bf);
void error(char *s,char *s2="");
char *arg[100];

#include "graph.h"

ofstream *initialise(char *[]);
void random_generate();

main(int argc, char *argv[])
{

    random_generate();
}

#define MAX_DIVISIONS   10
#define MAX_SAMPLES     10

void random_generate()
{
double fp;
unsigned d,bf;

    fp=0.1;d=5;bf=3;
    for (fp=0.1;fp<=0.5;fp += 0.02){
        get_min_E(fp,d,bf);
    }
    cout<<"\n\n"<<flush;
```

```
}

double get_min_E(double fp,unsigned d,unsigned bf)
{
unsigned M[10];
double e,p1,p2;
unsigned i,j;
Node *n;
double del;
double s;
double E=1;

    del=s=0;
    del = (1-fp)/MAX_DIVISIONS;
    M[0]=1;M[1]=12;M[2]=8;M[3]=4;M[4]=4;M[5]=4;M[6]=4;
    for (i=0;(i<MAX_DIVISIONS-1) ;i++) {
        s += del; e=0;
        for (j=0;j<MAX_SAMPLES;j++){
            PFG G;
            srand(rand() % 1000 + 1);
            n=G.generate(M,d,bf,0,fp,s,1-fp-s,NULL,0);
            G.calc_depth(G.get_start(),0);
            G.cluster_PFG(G.get_start());
            p1 = G.N_f();
            G.transform(G.get_start());
            p2 = G.N_f();
            e = e + (p1-p2)/(p1);
            cout<<"\n"<<s<<" "<<(double)(p1-p2)/(p1);
        }
        e = e/MAX_SAMPLES;
        if (E>e) {
            E = e;
        }
    }
return E;
}


void error(char *s,char *s2 )
{
    cerr << s << " " << s2 << "\n "<<flush;
}
```

```
/*
generates values for fp=0.1 ... 0.5 and cp=(1-fp)/2,sp=(1-fp)/2
*/

#include <iostream.h>
#include <fstream.h>
#include <libc.h>

#define TRUE      1
#define FALSE     0

extern "C" {
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
}
double get_min_E1(double fp,unsigned d,unsigned bf);
void error(char *s,char *s2="");
char *arg[100];

#include "graph.h"

ofstream *initialise(char *[]);
void random_generate();

main(int argc, char *argv[])
{

    random_generate();
}


#define MAX_DIVISIONS  10
#define MAX_SAMPLES    20

void random_generate()
{
double fp;
unsigned d,bf;

    fp=0.1;d=5;bf=3;
    for (fp=0.1;fp<=0.5;fp += 0.01){
        get_min_E1(fp,d,bf);
    }
    cout<<"\n\n"<<flush;
```

```
}


double get_min_E1(double fp,unsigned d,unsigned bf)
{
unsigned M[10];
double e,p1,p2;
unsigned j;
double E=1;
    e=0;
    M[0]=1;M[1]=8;M[2]=6;M[3]=4;M[4]=4;M[5]=4;M[6]=4;
        for (j=0;j<MAX_SAMPLES;j++){
            PFG G;
            srand(rand() % 1000 + 1);
            G.generate(M,d,bf,0,fp,(1-fp)/2,(1-fp)/2,NULL,0);
            G.calc_depth(G.get_start(),0);
            G.cluster_PFG(G.get_start());
            p1 = G.N_f();
            G.transform(G.get_start());
            p2 = G.N_f();
            e = e + (p1-p2)/(p1);
            cout<<"\n"<<fp<<" "<<(double) (p1-p2)/(p1)<<flush;
        }
        E = e/MAX_SAMPLES;
return E;
}


ofstream *initialise(char *argv[])
{
ofstream *fout;
    fout = new ofstream(argv[1]);
    return fout;
}


void error(char *s,char *s2 )
{
    cerr << s << " " << s2 << "\n "<<flush;
}
```

```
#include <iostream.h>
#include <fstream.h>
#include <libc.h>

#define TRUE        1
#define FALSE       0

extern "C" {
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
}
void error(char *s,char *s2="");
char *arg[100];

#include "graph.h"
#include "tree_node.h"
#include "traversal.h"

ofstream *initialise(int , char *[]);

main(int argc, char *argv[])
{
ofstream *fout;
int ng; /* Number of graphs to be analised */
    if (argc==1) {
        cout<<"\n Specify Input and Output files\n"<<flush;
        exit(0);
    }

    if ((argc-1)==1) ng =1; else ng =argc-2;
    fout = initialise(argc,argv);

        PFGS graphs(ng,arg,fout);

        graphs.release_time_static(ng);
        graphs.release_time(ng);
        graphs.print();
    *fout<<"\n\n";fout->close();
    delete fout;
}


ofstream *initialise(int argc,char *argv[])
```

```
{
ofstream *fout;
    arg[0] = new char[25];

    switch (argc) {
    case 1:
        sprintf(arg[0],"Input");
        fout = new ofstream("Output");
        break;
    case 2:
        strcpy(arg[0],argv[1]);
        fout = new ofstream("Output");
        break;
    default:
        strcpy(arg[0],argv[1]);
        for (int i=2;i<argc;i++){
            arg[i-1] =new char[25];
            strcpy(arg[i-1],argv[i]);
        }
        fout = new ofstream(argv[argc-1]);
    }
    return fout;
}

void error(char *s,char *s2 )
{
    cerr << s << " " << s2 << "\n "<<flush;
}
```

```cpp
#include <iostream.h>
#include <fstream.h>
#include <libc.h>

#define TRUE    1
#define FALSE     0

extern "C" {
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
}
void error(char *s,char *s2="");
char *arg[100];

#include "graph.h"

ofstream *initialise(char *[]);
void random_generate(ofstream *fout);

main(int argc, char *argv[])
{
ofstream *fout;
char inf[25];

    if (argc!=3) {
        cout<<"\n"<<argv[0]<<
        ":Need to specify Input and Output files \n"<<flush;
        exit(0);
    }
    strcpy(inf,argv[1]);
    fout = initialise(argv);
    PFG G(inf,fout);

    G.read_PFG(NULL);
    G.calc_depth(G.get_start(),0);
    G.cluster_PFG(G.get_start());
    G.transform(G.get_start());
    G.printf(G.get_start());

    fout->close();
    if (fout!=NULL) delete fout;
}
```

```
ofstream *initialise(char *argv[])
{
ofstream *fout;
    fout = new ofstream(argv[2]);
    return fout;
}

void error(char *s,char *s2 )
{
    cerr << s << " " << s2 << "\n "<<flush;
}
```

Bibliography

1. Stoyenko, A. D., Marlowe, T. J., "Polynomial-Time Transformations and Schedulability Analysis of Parallel Real-Time Programs with Restricted Resource Contention." *Research Report CIS 91-18.*, NJIT. To appear in Real-Time Systems in 1992.

2. Shaw, A. C., "Reasoning About Time in High-Level Language Software." *IEEE Transactions on Software Engineering*, pp. 875-889, SE-15, No. 7, July 1979.

3. Joachim Roos, "A Real-Time Support Processor for ADA Tasking." *ASPLOS-III Proceedings ACM*, April 3-6, 1989.

4. Kligerman, E., Stoyenko, A. D., "Real-Time Euclid: A Language for Reliable Real-Time Systems." *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 9, pp. 940-949.

5. Wolfgang A. Halang, Stoyenko, A. D., Constructing Predictable Real Time Systems, *Kluwer Academic Publishers*, 1991.

6. Leinbaugh, D. W., "Guaranteed Response Times in a Distributed Hard Real Time Environment." *IEEE Transactions on Software Engineering*, Vol SE-6, No 1, pp. 85-91, Jan 1980.

7. Alan Burns, Andy Wellings, Real Time Systems and Their Programming Languages, *Addison-Wesley Publishing Company*, 1989

8. Ullman, J., "Polynomial complete scheduling problems." *Proceedings 4th Symposium on OS Principles*, pages 96-101.

9. C. Liu and J. Layland, "Scheduling algorithms for multiprogramming in hard real time environment." *Journal of the ACM*, 20:46-61, 1973.

10. John A. Stankovic and Krithi Ramamritham, "Tutorial: Hard real time systems." *Computer society press of the IEEE*, 1988, catalog number EH0276-6.

11. H. Kobayashi, Modeling and Analysis: An introduction to systems performance evaluation methodology. *Addison-Wesley Publishing*, 1978.