Scheduling Processes with Release Times, Deadlines, Precedence, and Exclusion Relations

JIA XU AND DAVID LORGE PARNAS

Abstract—**We present an algorithm that finds an optimal schedule on a single processor for a given set of processes such that each process starts executing after its release time and completes its computation before its deadline, and a given set of precedence relations and a given set of exclusion relations defined on ordered pairs of process segments are satisfied. This algorithm can be applied to the important and previously unsolved problem of automated pre-run-time scheduling of processes with arbitrary precedence and exclusion relations in hard-real-time systems.**

Index Terms—**Automated pre-run-time scheduler, deadlines, exclusion, hard-real-time systems, precedence, scheduling algorithms.**

I. Introduction

We present an algorithm for solving the following problem: we are given a set of processes, where each process consists of a sequence of segments. Each segment is required to precede a given set of other segments. Each segment also excludes a given set of other segments, i.e., once a segment has started its computation it cannot be preempted by any segment in the set that it excludes. For each process, we are given a release time, a computation time, and a deadline. It is also assumed that we know the computation time and start time of each segment relative to the beginning of the process containing that segment.

Our problem is to find a schedule on a single processor for the given set of processes such that each process starts executing after its release time and completes its computation before its deadline, and all the precedence and exclusion relations on segments are satisfied.

Note that if we can solve the problem stated above, then we can also solve the special case where the release times and deadlines of each process are periodic, by solving the above problem for the set of processes occurring within a time period that is equal to the least common multiple of the periods of the given set of processes.

The algorithm presented here was designed to be used by a pre-run-time scheduler for scheduling processes with arbitrary precedence and exclusion relations in hard-real-time systems [3]. In such systems, precedence relations may exist between process segments when some process segments require information that is produced by other process segments. Exclusion relations may exist between process segments when some process segments must exclude interruption by other process segments to prevent errors caused by simultaneous access to shared resources, such as data, I/O devices, etc.

It has been observed that in many hard-real-time applications, the bulk of the computation can be confined to periodic processes where the sequencing and timing constraints are known in advance. That is, the release times and deadlines of processes besides the precedence and exclusion relations defined on them are known in advance. General techniques also exist for transforming a set of asynchronous processes into an equivalent set of periodic processes [16], [17]. Thus it is possible to use a pre-run-time scheduler to make scheduling decisions before run time. Pre-run-time scheduling has many advantages compared to run time scheduling: precious run time resources required for run time scheduling and context switching can be greatly reduced, and more importantly, it is easier to guarantee in advance that real-time deadlines will be met.

However, up to the present time, the automated pre-run-time scheduler for processes with arbitrary precedence and exclusion relations has remained "an unsolved problem" [3]. As will be discussed below, no algorithm previously existed for solving the problem of finding an optimal schedule for a set of processes with arbitrary release times, deadlines, precedence and exclusion relations. In the past, designers of safety-critical hard-real-time systems have had to resort to ad hoc methods and perform pre-run-time scheduling by hand. Except for very simple problems, ad hoc and manual methods are prone to errors, time consuming, and they often fail to find a feasible schedule even when one exists.

The algorithm presented here makes it possible to completely automate the task of pre-run-time scheduling processes with arbitrary precedence and exclusion relations. Currently we are working on producing a practical system that uses this algorithm to systematically search for a feasible schedule when given a set of release time, deadline, precedence, and exclusion relation parameters. Such a system would greatly facilitate the task of pre-run-time scheduling. It would virtually eliminate any possibility of errors in the computation of schedules. Not only would it

Manuscript received April 3, 1989; revised July 1, 1989. Recommended by P. A. Ng. This work was supported in part by Natural Sciences and Engineering Research Council of Canada operating grants to D. L. Parnas and K. C. Sevcik.

J. Xu is with the Department of Computer Science, York University, 4700 Keele Street, North York, Ont. M3J 1P3, Canada.

D. L. Parnas is with the Department of Computing and Information Sciences, Queens University, Kingston, Ont. K7L 3N6, Canada.

IEEE Log Number 8933204.

be capable of finding a feasible schedule whenever one exists, it would also be capable of informing the user whenever no feasible schedule exists for a given set of parameters much faster and reliably than any ad hoc or manual method. In the latter case, it could also provide the user with useful information on which parameters should be modified in order to obtain a feasible schedule. Such a system would be particularly useful for applications in which changes in the system often occur and schedules have to be frequently recomputed.

In [16], Mok treats in detail techniques which allow one to use a pre-run-time scheduler to make scheduling decisions before run time for both periodic and synchronous processes by replacing asynchronous processes with an equivalent set of periodic processes. Extensive surveys of scheduling problems and algorithms can be found in [2], [8], and [10]. For solving the problem of finding a feasible schedule for a set of processes where each process must execute between a given release time and deadline, all previously reported algorithms either solve the special case where each process consists of a single segment that does not allow preemptions, or, solve the special case where each process consists of a single segment that can be preempted by any other process. The latter case can be solved in polynomial time, even if n processors are used [11], [13]. In the former case, the problem is NP-complete in the strong sense, even if only one processor is used [6], which effectively excludes the possibility of the existence of a polynomial time algorithm for solving the problem. For special cases where all processes have unit computation time, and no preemptions are allowed, polynomial time algorithms have been obtained [4], [5], [7], [18]. Several heuristics have also been proposed or studied for the former case [12], [9]. For solving the case where each process consists of a single segment that does not allow preemptions, and a single processor is used, an elegant implicit enumeration algorithm was presented in [14]. Another implicit enumeration algorithm of comparable efficiency is described in [1].

We do not know of any published algorithm that solves the more general problem where some portions of a process are preemptable by certain portions of other processes, while other portions of a process are not preemptable by certain portions of other processes. Such problems occur frequently in many real world situations. Since the major concern in a hard-real-time environment is meeting deadlines, none of the previously published algorithms were applicable to our problem, since assuming all processes are completely preemptable would allow simultaneous access to shared resources which could have disastrous consequences; whereas assuming all processes are completely nonpreemptable would seriously affect our ability to meet deadlines.

The problem as stated above can easily be proved to be NP-hard (even the special case where each process is composed of a single segment that excludes all other single segment processes is NP-hard). The objective of the work reported here was to find a feasible schedule whenever one exists for a given set of problem parameters. This requirement together with the fact that the problem to be solved is NP-hard, effectively excludes all other types of solutions except solutions that implicitly enumerates all possible feasible schedules.

Although it is possible to construct pathological problem instances where the algorithm would require an amount of computation time that is exponentially related to the problem size, it is extremely unlikely that such pathological problem instances would occur in practical hard-real-time system applications. Our experience has shown that even with difficult problems of very large size, the algorithm can still provide an optimal solution within reasonable time.

One can easily see that our algorithm is also applicable to a wide range of practical problems that are not directly related to the field of computer science. Although we have adopted the terminology commonly used in computer science, readers familiar with the terminology of operations research may substitute the terms "job" or "task" for "process," "machine" for "processor," "processing time" for "computation time," and "portions of a job that cannot be interrupted by portions of other jobs" for "segments that exclude other segments."

A very useful property of this algorithm is that at each intermediate stage of the algorithm a complete schedule is constructed. At the beginning, the algorithm starts with a schedule that is obtained by using an earliest-deadline-first strategy. Then it systematically improves on that initial schedule until an optimal or feasible schedule is found. Thus, even if we have to terminate the algorithm prematurely, it would still provide a complete schedule that is at least as good as any schedule obtained by using an earliest-deadline-first heuristic. Schedules obtained by using an earliest-deadline-first heuristic have the best known upperbound on lateness among all previously proposed heuristics for scheduling nonpreemptable process with arbitrary release times and deadlines [9]. The earliest-deadline-first strategy is also optimal for scheduling processes that are completely preemptable [15]. Under any circumstance, for solving the problem of scheduling processes with arbitrary release times, deadlines, and precedence and exclusion relations defined on process segments, this algorithm should outperform any previously proposed heuristic.

In the next section, we provide an overview of the algorithm. Basic notation and definitions are introduced in Section III. In Section IV we show how to improve on a valid initial solution. In Section V we describe the strategy used to search for an optimal or feasible solution. The empirical behavior of the algorithm is described in Section VI. Finally, conclusions are presented in Section VII.

II. Overview of the Algorithm

From the computation time and start time of each segment relative to the beginning of the process containing that segment, and the release time, computation time, and deadline of each process, one should be able to compute the release time, computation time, and the deadline for each segment.

Our algorithm finds a valid schedule in which the lateness of all segments in the schedule is minimized, while satisfying a given set of "*EXCLUDE*" relations and a given set of "*PRECEDE*" relations defined on ordered pairs of segments. The set of *EXCLUDE* relations and the set of *PRECEDE* relations are initialized to be identical with those exclusion and precedence relations required in our original problem.

If the minimum lateness of all schedules is greater than zero, then no feasible schedule exists that will satisfy all deadline constraints. Otherwise, the algorithm will find a feasible schedule that meets all deadline constraints.

Our algorithm uses a branch-and-bound technique. It has a search tree where at its root node we use an earliest-deadline-first strategy to compute a schedule called a "valid initial solution" that satisfies the release time constraints and all the initial *EXCLUDE* and *PRECEDE* relations.

At each node in the search tree, we find the latest segment in the valid initial solution computed at that node. We identify two "expand" sets of segments G_1 and G_2 such that the valid initial solution can be improved on if either the latest segment is scheduled before a segment in the expand set G_1; or, the latest segment preempts a segment in the expand set G_2.

For each segment in the expand sets G_1 and G_2, we create a successor node in which we add appropriate *PRECEDE* or "*PREEMPT*" relations, such that if a valid initial solution for the successor node is computed using those new additional relations, then the latest segment in the parent node would be scheduled before a segment in G_1, or preempt a segment in G_2 whenever possible.

For each node in the search tree, we also compute a lower bound on the lateness of any schedule leading from that node. The node that has the least lower bound among all unexpanded nodes is considered to be the node that is most likely to lead to an optimal solution—we always branch from the node that has the least lower bound among all unexpanded nodes. In case of ties, we choose the node with least lateness among the nodes with least lower bound.

We continue to create new nodes in the search tree until we either find a feasible solution, or, until there exists no unexpanded node that has a lower bound less than the least lateness of all valid initial solutions found so far. In the latter case, the valid initial solution that has the least lateness is an optimal solution.

The ways in which we use *PRECEDE* and *PREEMPT* relations to either schedule the latest segment before a segment in the expand set G_1 or let the latest segment preempt a segment in the expand set G_2 cover *all* possible ways of improving on a valid initial solution. This guarantees that in the latter case, the solution is globally optimal rather than locally optimal.

In the following section, we shall formally define all the terms mentioned above.

III. Notation and Definitions

In order to solve the problem stated above, we first introduce the following definitions and notations.

Let the *set of processes* be denoted by P.

Each process $p \in P$ consists of a finite sequence of *segments* $p[0], p[1], \cdots, p[n[p]]$, where $p[0]$ is the first segment and $p[n[p]]$ is the last segment in process p.

For each segment i, we define:

- a release time $r[i]$;
- a deadline $d[i]$;
- a computation time $c[i]$;

It is assumed that $r[i]$, $d[i]$, and $c[i]$ have integer values.

Let the *set of all segments* belonging to processes in P be denoted by $S(P)$. Each segment i consists of a sequence of *segment units* $(i, 0), (i, 1), \cdots, (i, c[i] - 1)$, where $(i, 0)$ is the first segment unit and $(i, c[i] - 1)$ is the last segment unit in segment i.

We define the *set of segment units of* $S(P)$:

$$U = \{(i, k) \mid i \in S(P) \land 0 \le k \le c[i] - 1\}.$$

Intuitively, a segment unit is the smallest indivisible granule of a process. Each segment unit requires unit time to execute, during which it cannot be preempted by any other process. The total number of segment units in each segment is equal to the computation time required by that segment.

A *schedule* of a set of processes P is a total function π: $U \to [0, \infty)$ satisfying the following properties:

1) $\forall t \in [0, \infty): |\{(i, k) \in U \mid \pi(i, k) = t\}| \le 1.$

2) $\forall (i, k_1), (i, k_2) \in U: (k_1 < k_2) \Rightarrow (\pi(i, k_1) < \pi(i, k_2)).$

3) $\forall p, i, j, p \in P, 0 \le i, j \le n[p]$:
$$(i < j) \Rightarrow (\pi(p[i], c[p[i]] - 1) < \pi(p[j], 0))$$

Above, condition 1) states that no more than one segment can be executing at any time. Condition 2) states that a schedule must preserve the ordering of the segment units in each segment. Condition 3) states that a schedule must preserve the ordering of the segments in each process.

We say *segment* i *executes at time* t iff $\exists k, 0 \le k \le c[i] - 1: \pi(i, k) = t$.

We say *segment* i *executes from* t_1 *to* t_2 iff $\exists k, \forall t, 0 \le k \le c[i] - 1, 0 \le t \le t_2 - t_1 - 1: \pi(i, k + t) = t_1 + t$.

We define the *start time* of segment i to be $s[i] = \pi(i, 0)$;

We define the *completion time* of segment i to be $e[i] = \pi(i, c[i] - 1) + 1$.

The *lateness of a segment* i in a schedule of P is defined by $e[i] - d[i]$.

The *lateness of a schedule of* P is defined by $\max\{e[i] - d[i] \mid i \in S(P)\}$.

We define a *latest segment* to be a segment that realizes the value of the lateness of the schedule.

We introduce the *PRECEDE* relation and *EXCLUDE* relation on ordered pairs of segments together with the notion of a "valid schedule."

A *valid schedule* of a set of processes P is a schedule of P satisfying the following properties.

$\forall i, j \in S(P)$:

1) $s[i] \geq r[i]$

2) $(i \textit{ PRECEDES } j) \Rightarrow (e[i] \leq s[j])$

3) $(i \textit{ EXCLUDES } j \wedge s[i] < s[j])$
$\Rightarrow (e[i] \leq s[j])$

Above, condition 1) states that each process can only start execution after its release time. Condition 2) states that in a valid schedule, if segment i *PRECEDES* segment j, then under all circumstances, segment j cannot start execution before segment i has completed its computation. Condition 3) states that in a valid schedule, if segment i *EXCLUDES* segment j, then segment j is not allowed to preempt segment i. That is, if segment i started execution before segment j, then segment j can only start execution after segment i has completed its computation.

We initialize the set of *PRECEDE* relations and the set of *EXCLUDE* relations to be identical with the precedence and exclusion relations that must be satisfied in the original problem. In addition, in order to enforce the proper ordering of segments within each process, we let $p[k]$ *PRECEDE* $p[k+1]$ for all $p \in P$, and for all k, $0 \leq k \leq n[p] - 2$. Thus, a valid schedule would satisfy all the release time, exclusion, and precedence constraints in the original problem.

A *feasible schedule* of a set of processes P is a valid schedule of P such that its lateness is less than or equal to zero.

An *optimal schedule* of a set of processes P is a valid schedule of P with minimal lateness.

The *adjusted release time* $r'[i]$ of segment i is defined by

1) $r'[i] = r[i]$, *if* $\nexists j : j \textit{ PRECEDES } i$;

else

2) $r'[i] = \max\{r[i], r'[j] + c[j] \mid j \textit{ PRECEDES } i\}$

At any time t, $t \in [0, \infty]$, we say segment "*j is ELIGIBLE at t*" iff:

1) $t \geq r'[j] \wedge \neg(e[j] \leq t)$

2) $\nexists i : i \textit{ PRECEDES } j \wedge \neg(e[i] \leq t)$

3) $\nexists i : i \textit{ EXCLUDES } j \wedge s[i] < t \wedge \neg(e[i] \leq t)$

The above definition guarantees that at any time t, if segment j *is ELIGIBLE at* t, then j can be put into execution at t, while satisfying all the properties of a valid schedule.

We also introduce a third relation that will be used in our algorithm—the *PREEMPT* relation on pairs of segments together with the notion of a "valid initial solution."

A *valid initial solution* for a set of processes P is a valid schedule of P satisfying the following properties:

$\forall t \in [0, \infty)$:

1) $\forall j : (\exists i : ((i \textit{ PREEMPTS } j \wedge i \textit{ is ELIGIBLE at } t)$
$\vee (d[i] < d[j] \wedge \neg(j \textit{ PREEMPTS } i)$
$\wedge\, i \textit{ is ELIGIBLE at } t)$
$\vee (d[i] = d[j] \wedge c[i] > c[j]$
$\wedge \neg(j \textit{ PREEMPTS } i) \wedge i \textit{ is ELIGIBLE at } t))$
$\Rightarrow \neg(j \textit{ executes at } t))$

2) $\exists i$: *i is ELIGIBLE at t*
$\Rightarrow \exists i : i \textit{ executes at } t$

Above, condition 1) states that in a valid initial solution, if at least one segment i *PREEMPTS* segment j and i is *ELIGIBLE* at time t; or if at least one segment i has a shorter deadline than j and i is *ELIGIBLE* at time t and j does NOT *PREEMPT* i; or if at least one segment i has the same deadline but a longer computation time than j and i is *ELIGIBLE* at time t and j does NOT *PREEMPT* i, then segment j cannot execute at time t. Condition 2) states that in a valid initial solution, at any time t, if at least one segment is *ELIGIBLE*, then one segment should execute at time t. Condition 2) effectively guarantees that all segments will eventually be completed in a valid initial solution, provided that all relations on segments are "consistent" as defined below.

We define each pair of relations on segments indicated by an "x" in the following table to be *inconsistent*. All other pairs of relations on segments are *consistent*.

	i PC j	j PC i	i EX j	j EX i	i PM j	j PM i
i PC j:		x				x
i EX j:						x
i PM j:		x		x		x

i PC j : i *PRECEDES* j
i EX j : i *EXCLUDES* j
i PM j : i *PREEMPTS* j
x : inconsistent

In addition to satisfying release time, exclusion and precedence constraints, a valid initial solution also satisfies execution priority constraints defined by the set of *PREEMPT* relations and deadlines.

Initially, we set the set of *PREEMPT* relations to be empty. New *PREEMPT* relations as well as new *PRECEDE* relations will be defined and used by the algorithm to reschedule the latest segment earlier in order to improve on existing valid initial solutions.

The following (simplified) procedure uses an earliest-deadline-first strategy to compute a valid initial solution in which release time constraints and a given set of *EXCLUDE, PRECEDE*, and *PREEMPT* relations are enforced:

```
$t \leftarrow 0$
while $\neg (\forall i : e[i] \le t)$ do
  begin
    if $(\exists i : t = r'[i] \vee t = e[i])$ then
      begin
        Among the set
        $\{ j \mid j$ is ELIGIBLE at $t$
          $\wedge (\nexists i : i$ is ELIGIBLE at $t \wedge i$ PREEMPTS $j)$
        $\}$
        select the segment $j$ that has min $d[j]$.
        in case of ties, select the segment $j$ that has
          max $c[j]$.
        put $j$ into execution.
      end
    $t \leftarrow t + 1$
  end
```

A more detailed implementation of the procedure for computing a valid initial solution can be found in Appendix 1.

See Examples 1–5 in Appendix 3 for examples of schedules corresponding to valid initial solutions.

IV. How to Improve on a Valid Initial Solution

Let j be the latest segment in a valid initial solution. (If there exists more than one segment that have maximum lateness, then let j be the segment that completed last among those segments.)

Any nonoptimal schedule may be improved on only if j can be rescheduled earlier.

We define the set of segments $Z[i]$ recursively as follows:

1) $i \in Z[i]$;

2) $\forall k$:

if $\exists l, l \in Z[i]$:

$$\left(e[k] = s[l] \wedge \left(\exists l', l' \in Z[i] : r'[l'] < e[k]\right) \vee \left(s[l] < e[k] < e[i]\right)\right.$$

then $k \in Z[i]$

The properties of a valid initial solution imply that in any schedule that corresponds to a valid initial solution:

- $Z[i]$ is the set of segments that precede (and include) i in a period of continuous utilization of the processor;
- $e[i]$ is the earliest possible completion time for the entire set of segments $Z[i]$.
- any nonoptimal schedule may be improved on only by scheduling some segment $k \in Z[j]$ such that $d[j] < d[k]$ later than the latest segment j.

As an example, in the valid initial solution of the root node of the search tree of Example 5: $D \in Z[D]$ from 1); $A \in Z[D]$ because $e[A] = s[D] \wedge r'[D] < e[A]$; $B, C \in Z[D]$ because $s[A] < e[B], e[C] < e[D]$. Thus $Z[D] = \{A, B, C, D\}$. $e[D]$ is the earliest possible completion time for the entire set of segments $Z[D]$—if any other order for the segments in $Z[D]$ is chosen, the last segment in that new order cannot complete before $e[D]$.

We define two *expand sets* G_1 and G_2 as follows:

$$\begin{aligned} G_1 = \{ i \mid\ & i \in Z[j] \wedge d[j] < d[i] \\ & \wedge\ i\ \textit{EXCLUDES}\ j \\ & \wedge \neg (i\ \textit{PRECEDES}\ j) \\ & \wedge \neg (i\ \textit{PREEMPTS}\ j) \} \end{aligned}$$

$$\begin{aligned} G_2 = \{ i \mid\ & i \in Z[j] \wedge d[j] < d[i] \\ & \wedge \neg (i\ \textit{EXCLUDES}\ j) \\ & \wedge \neg (i\ \textit{PRECEDES}\ j \\ & \wedge \neg (i\ \textit{PREEMPTS}\ j) \\ & \wedge \nexists l : (\exists k, t : 0 \le k \le c[l] - 1, 0 \\ & \le t < \infty : s[i] \le \pi(l, k) \le e[j]) \\ & \quad \text{\% an execution of } l \text{ occurs between } i \text{ and } j \text{ \%} \\ & \quad \wedge (i\ \textit{PRECEDES}\ l \vee i\ \textit{PREEMPTS}\ l) \} \end{aligned}$$

G_1 is the set of segments that, if scheduled after j, may reduce the maximum lateness.

G_2 is the set of segments that, if preempted by j, may reduce the maximum lateness.

As examples, the valid initial solution of the root node of the search tree in Example 1 can be improved on by scheduling $A \in G_1$ after the latest segment C. The valid initial solution of the root node of the search tree in Example 4 can be improved on if the latest segment E preempts $A \in G_2$. In Example 4 $B \notin G_2$ because there exists D such that B *PRECEDES* D and an execution of D occurs between B and E.

By making use of the fact that $e[i]$ is the earliest possible completion time of the entire set of segments $Z[i]$, we can compute a lower bound on the lateness of any valid initial solution satisfying a given set of *EXCLUDE, PRECEDE*, and *PREEMPT* relations with the following formula:

$$\begin{aligned} \text{let } K[i] = \{ k \mid\ & k \in Z[i] \wedge k \ne i \wedge d[i] \\ & < d[k] \wedge \neg (k\ \textit{PRECEDES}\ i) \\ & \wedge \neg (k\ \textit{PREEMPTS}\ i) \} \end{aligned}$$

if $K[i] = \varnothing$

then $LB[i] = e[i] - d[i]$

else $LB[i] = e[i] + \min \{ GAP[k, i] - d[k] \mid k \in K[i] \}$ where

if $\neg (k\ \textit{EXCLUDES}\ i)$ then $GAP[k, i] = 0$

$$\text{else } GAP[k, i] = \max\Big\{0, -s[k] + \min\Big\{r'[l] \,\Big|\, l \in Z[i] \wedge k \neq l \wedge s[k] < s[l] \leq s[i] \wedge \neg(k \; PRECEDE \; l)\Big\}\Big\}$$

$$LB_1[i] = \min\{LB[i], e[i] - d[i]\}$$

$$LB_2[i] = r'[i] + c[i] - d[i]$$

$$\text{lowerbound} = \max\{LB_1[i], LB_2[i] \mid i \in S(P)\}$$

The lower bound function can be derived by observing the following: if the set $K[i]$ is empty then the lateness of i, i.e., $e[i] - d[i]$ cannot be improved on. This is because if any other segment $k \in Z[i]$ where $d[k] \leq d[i]$ is scheduled last, then k would be at least as late as the lateness of i. If k *EXCLUDEs* i and $s[k] < \min\{r'[l] \mid s[k] < s[l] \leq s[i]\}$, then from the properties of a valid initial solution, scheduling k after i would leave a gap in the new schedule that starts at $s[k]$ and ends at $\min\{r'[l] \mid s[k] < s[l] \leq s[i]\}$. (Note that l could be equal to i), and the lateness of the new schedule would be at least $e[i] - d[k]$ plus the gap size. $LB_2[i]$ is a trivial lowerbound on the lateness of any segment i.

V. Searching for an Optimal or Feasible Solution

We now define a search tree that has as its root node the valid initial solution that satisfies all the *EXCLUDE* and *PRECEDE* relations in the original problem specification.

At each node in the search tree we compute the lower bound and two expand sets G_1 and G_2. Let segment j be the latest segment in the valid initial solution computed at that node.

For each segment $k \in G_1$, we create a successor node that corresponds to a new problem, in which we assign a new relation j *PRECEDES* k. If we apply the procedure above and compute a new valid initial solution in which the new relations are enforced, then segment k will be scheduled later than segment j in the new schedule.

For each segment $k \in G_2$, we create a successor node that corresponds to a new problem, in which for all segments l such that k *EXCLUDES* l and an execution of l occurs between k and j, we assign the relation l *PRECEDES* k, and for all segments q such that k does NOT *EXCLUDE* q and an execution of q occurs between k and j, we assign the relation q *PREEMPTS* k and the relation j *PREEMPTS* k. We let each successor node inherit all relations assigned to any of its predecessor nodes. If we apply the procedure above and compute a new valid initial solution in which the new relations are enforced, then segment k will be preempted by segment j in the new schedule if possible. After generating the valid initial solution for each new successor node, we test it for optimality. If the optimal solution is not discovered among any of the resulting problems, then we proceed to create new successor nodes in a similar manner. We use a strategy of branching from the node with the least lower bound. In case of ties, we choose the node with least lateness among the nodes with least lower bound.

The steps of the algorithm are as follows:

(For a more detailed implementation of the algorithm see Appendix 2.)

Step 0: Compute an initial valid solution and the corresponding lowerbound. Find the latest segment j and its lateness. If its lateness equals its lowerbound then stop—the schedule is optimal. Otherwise, call the node corresponding to the schedule of the parent node.

Step 1: Find the expand sets G_1 and G_2 and create $|G_1| + |G_2|$ new child nodes. For each node corresponding to a segment k in G_1, assign a new relation j *PRECEDES* k. For each node corresponding to a segment k in G_2, for all segments l such that k *EXCLUDES* l and an execution of l occurs between k and j, assign a new relation l *PRECEDES* k, and, for all segments q such that k does NOT *EXCLUDE* q and an execution of q occurs between k and j, assign the relation q *PREEMPTS* k and the new relation j *PREEMPTS* k.

Let each child node inherit all relations assigned to any of its predecessor nodes.

Recompute a valid initial solution, lowerbound and find the latest segment and its lateness for each child node.

Step 2: If Steps 3 and 4 have been performed for all child nodes then close the parent node and go to Step 5.

Otherwise, select the child node with the least lateness.

Step 3: Set minlateness ← min {minlateness, lateness (childnode)}.

If minlateness is less than or equal to the least lowerbound of all open nodes then **stop**—the solution is optimal.

Step 4: If lateness (childnode) = lowerbound (childnode) then close this child node and return to step 2—this solution is locally optimal.

If minlateness is less than lowerbound (childnode) then close this childnode—this node will never lead to a solution that is better than the current minlateness.

Return to step 2.

Step 5: Select among all open nodes the node with the least lower bound, in case of ties, select the node with least lateness. Call this node the parent node and goto step 1. □

(See Examples 1–5 in Appendix 3.)

If a feasible schedule is considered sufficient, to achieve more efficiency, instead of terminating the algorithm only when a minimum lateness schedule has been found, one may terminate the algorithm as soon as a feasible schedule in which all deadlines are met is found. One could also adopt a strategy of terminating the search whenever a schedule has been found such that its lateness is within a prespecified ratio of optimal. An upperbound on that ratio can be computed with the formule (lateness $- L)/L$ where L is the least lower bound of all nodes belonging to the open node set.

VI. Empirical Behavior of the Algorithm

We have written a program in Pascal that implements the algorithm described above.

Observation of the empirical behavior of the algorithm indicated that this algorithm consistently generated significantly fewer nodes than one of the best algorithms reported so far that solves the special case where each process consists of only one segment that excludes all other segments [14].

We restricted ourselves to comparing the number of nodes generated for an identical problem sample, because this is the major factor that determines the size of the problem that can be effectively computed—it is basically this number that will grow exponentially when the problem size increases.

By comparing the two algorithms on sample problems corresponding to the special case where each process consists of only one segment that excludes all other segments, we found that for problem sizes of 25 (number of segments), our algorithm frequently generated 25% fewer nodes than the algorithm reported in [14]. When the problem size doubled to 50, our algorithm frequently generated 44% (approximately $1 - (1 - 0.25)^2$) fewer nodes. When we doubled the problem size again, the difference became even greater—their algorithm was unable to terminate after generating several tens of thousands of nodes, while our algorithm terminated on the same problem sample after generating only a few thousand nodes. It was also observed that for all problem samples of the general case (arbitrary exclusion relations defined on segments) that we constructed, solving them with our algorithm always generated fewer nodes before an optimal schedule was found than if all segments excluded each other (which corresponds to the special case).

Thus the performance of our algorithm on the general case in terms of the number of nodes generated should be much better than the performance reported in [14] when solving the special case.

VII. Conclusions

The major contribution of our algorithm is that it solves a very general and important problem that no other reported algorithm is capable of solving. It is the first algorithm that is able to systematically search for an optimal or feasible schedule that satisfies a given set of release time, deadline, precedence, and exclusion constraints defined on process segments. The algorithm can be applied to the important and previously unsolved problem of automated pre-run-time scheduling of processes with arbitrary precedence and exclusion relations in hard-real-time systems.

With our algorithm it is possible to take into account the cost of context switching. All we need to do is add to the computation time of each segment the following: 1) the time required to save the status of a preempted segment, 2) the time required to load a new segment, and 3) the time required to restart a preempted segment. This is because the only possible time where a process switch may take place is either at the adjusted release time or at the completion time of a segment. Furthermore, each segment can only preempt any other segment once. Hence we can always "charge" the cost of a context switch to the preempting segment so that all deadlines will be met. (See [15] for a similar argument for the earliest-deadline-first strategy.)

When implementing this algorithm, it may be advantageous to make space-time tradeoffs to match available resources. If our major constraint is space instead of time, we might consider only storing at each node partial information that is different from the information stored at its ancestor nodes, then whenever we need complete information to proceed at a certain node, we use the information stored at its ancestor nodes to reconstruct the complete information required at that node. For example, we only stored new *PRECEDE* and *PREEMPT* relations at each node when implementing our algorithm, which resulted in a significant saving of space without seriously affecting computation time.

One may also include an initial problem parameter verification stage that performs a preliminary analysis of all the initial problem parameters and modifies or rejects if necessary any problem parameters that are either redundant or inconsistent with other parameters prior to using this algorithm.

We note that this algorithm can be easily generalized to the case where exclusion regions within each process overlap or are embedded within each other.

For future work, we will explore ways of generalizing this algorithm to solve the problem of scheduling processes with release times, deadlines, precedence and exclusion relations on *n* processors. Another interesting direction for future work would be to explore ways of generalizing this algorithm to solve the problem with additional resource constraints [19].

Appendix I
An Implementation of the Procedure for Computing a Valid Initial Solution

The following procedure computes a valid initial solution in which the release time constraints and a set of *EXCLUDE*, *PRECEDE*, and *PREEMPT* relations are satisfied:

```
lastt := ⟨ any negative value ⟩;
lastseg := ⟨ any segment index ⟩;
idle := true;
for each segment i do
      begin
         started[ i ] := false;
         completed[ i ] := false;
         comptimeleft[ i ] := c[ i ];
         s[ i ] := -1;
      end;
   t := 0;
   while not( for all segments i: completed[ i ] = true ) do
   begin
      t := min { t | t > lastt and ((exists i: t = r'[ i ]) or
               ((idle = false) and (comptimeleft[ lastseg ] = t - lastt)))
               };
      if idle = false then
      begin
        % in the valid initial solution computed by the procedure: %
        let segment lastseg execute from lastt to t;
        comptimeleft[lastseg] := comptimeleft[lastseg] − (t −
        lastt);
        if comptimeleft[lastseg] = 0 then
          begin
            completed[lastseg] := true;
            e[lastseg] := t;
          end;
      end;
      S ← { j | j is ELIGIBLE and no other segment i exists
              such that i is also ELIGIBLE and i
              PREEMPTS j
      }
    if S is empty then idle := true
      else
        begin
          idle := false;
          S1 ← { j | d[j] = min { d[i] | i in S } }
          select segment x such that c[x] = max { c[i] | i in S1
          };
          if not started[x] then
            begin
              started[x] := true;
              s[x] := t;
            end;
          lastseg := x;
      end;
    lastt := t;
  end;
```

Above, "lastt" is the last time that the procedure tried to select a segment for execution. "lastseg" is the segment that was last selected for execution. "idle" indicates whether there was any segment selected at lastt. "started[i]" indicates whether segment i had started execution. "completed[i]" indicates whether segment i had completed execution. "comptimeleft[i]" is the remaining computation time of segment i.

Appendix 2
An Implementation of the Main Algorithm

```
begin {main}
  nodeindex := 0
  initialize(PC(nodeindex), EX)
  PM(nodeindex) ← ∅
  optimal := false;
  feasible := false;
  opennodeset ← ∅
  if consistent(PC(nodeindex), EX, PM(nodeindex)) then
  begin
    schedule(nodeindex) ← validinitialsolution(PC(nodeindex),
      EX, PM(nodeindex))
    leastlowerbound := lowerbound(nodeindex)
    if lateness(nodeindex) = least lowerbound then
      optimal := true;
    if lateness(nodeindex) ≤ 0 then
      feasible := true;
    if not (optimal or feasible) then
    begin
      opennodeset ← { nodeindex }
      minlateness := lateness(nodeindex);
      minlatenode := nodeindex;
      while not(optimal or feasible or spacetimelimitsexceeded)
        do
      begin
        lowestboundset ← {l | lowerbound(l) = leastlower-
          bound}
        select parentnode such that:
          lateness(parentnode) = min {lateness(i)|i ∈ low-
            estboundset}
        j := latestsegment(schedule(parentnode))
        firstchildnode := nodeindex + 1
        for each segment k ∈ G_1(parentnode)
          begin
            nodeindex := nodeindex + 1
            PC(nodeindex) ← PC(parentnode) ∪ {(j,k)}
          end
        for each k ∈ G_2(parentnode)
          begin
            nodeindex := nodeindex + 1
            PC(nodeindex) ← PC(parentnode)
            for all l such that:
              k EX l and an execution of l
              occurs between k and j in sched-
                ule(parentnode):
              begin
                PC(nodeindex) ← PC(nodeindex) ∪
                  {(l,k)}
              end
            PM(nodeindex) ← PM(parentnode) ∪ {(j,k}
            for all q such that:
              not(k EX q) and an execution of q
              occurs between k and j in sched-
                ule(parentnode):
              begin
                PM(nodeindex) ← PM(nodeindex) ∪
                  {(q,k}
              end
          end
        opennodeset ← opennodeset - {parentnode}
        if not (optimal or feasible) then
        for childnode := firstchildnode to nodeindex do
        begin
          if consistent(PC(childnode), EX, PM(childnode))
            then
          begin
            schedule(childnode) ←
              validinitialsolution(PC(childnode), EX,
                PM(childnode))
            if lateness(childnode) < minlateness then
            begin
              minlateness := lateness(childnode);
              minlatenode := childnode;
            end;
            if lateness(childnode) ≤ 0 then
              feasible := true
            else
              if minlateness > lowerbound(childnode) then
                opennodeset ← opennodeset ∪ {child-
                  node}
            end;
        end;
        leastlowerbound ← min{lowerbound(i) | i ∈ openno-
        deset}
        if opennodeset = ∅ or (minlateness ≤ leastlowerbound)
          then
            optimal := true;
      end;
      minlateschedule := schedule(minlatenode)
    end;
  end;
end.
(end of algorithm)
```

In the algorithm above, a node in the "opennodeset" is a node that does not have successors, but may be selected as the node to be branched from next. "PC(nodeindex)" and "PM(nodeindex)" are respectively the set of *PRECEDE* relations and the set of *PREEMPT* relations associated with the node identified by "nodeindex." "EX" is the (constant) set of *EXCLUDE* relations. "schedule(nodeindex)" is the valid initial solution computed using PC(nodeindex), EX and PM(nodeindex). "lateness(nodeindex)" is the lateness of schedule(nodeindex). "lowerbound(nodeindex), G_1(nodeindex), and G_2(nodeindex)" are, respectively, the lowerbound and the two expand sets computed from schedule(nodeindex).

To achieve more efficiency, instead of terminating the algorithm only when a minimum lateness schedule has been found, the algorithm terminates as soon as a feasible schedule in which all deadlines are met is found; or, when a predefined space/time limit is exceeded.

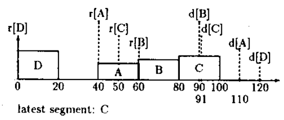
APPENDIX 3

EXAMPLES 1–5

Example 1.

r[A] = 40 r[B] = 60 r[C] = 50 r[D] = 0 **A EXCLUDES B C EXCLUDES A**
c[A] = 20 c[B] = 20 c[C] = 20 c[D] = 20 **B EXCLUDES A B EXCLUDES C**
d[A] = 110 d[B] = 90 d[C] = 91 d[D] = 120 **A EXCLUDES C C EXCLUDES B**

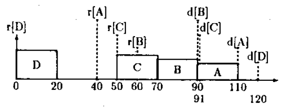
Root node of search tree:



latest segment: C
lateness = e[C] - d[C] = 100 - 91 = 9
Z[C] = {A, B, C}
lowerbound = LB[C] = e[C] + (r'[C] - s[A] - d[A]) = 100 + (50 - 40 - 110) = 0

G1 = {A} G2 = ∅

C PRECEDES A

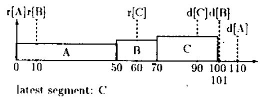


latest segment: A, B
lateness = e[A] - d[A] = e[B] - d[B] = 0
Z[A] = {C, B, A}
Z[B] = {C, B}
lowerbound = LB[A] = e[A] -d[A]
= LB[B] = e[B] - d[B] = 0
(globally optimal since lateness equal to
least lower bound of all open nodes)

Example 2.

r[A] = 0 r[B] = 10 r[C] = 60 **A EXCLUDES B**
c[A] = 50 c[B] = 20 c[C] = 30 **B EXCLUDES C**
d[A] = 110 d[B] = 101 d[C] = 90

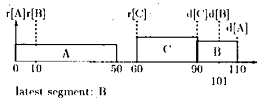
Root node of search tree:



latest segment: C
lateness = e[C] - d[C] = 100 - 90 = 10
Z[C] = {A, B, C}
lowerbound = LB2[C] = r'[C] + c[C] - d[C] = 60 + 30 - 90 = 0

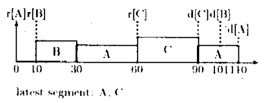
G1 = {B} G2 = {A}

C PRECEDES B



latest segment: B
lateness = e[B] - d[B] = 110 - 101 = 9
Z[B] = {C, B}
lowerbound = LB[B] = e[B] -d[B] = 110 -101 = 9
(locally optimal since lateness equal to
lower bound - close this node)

C PREEMPTS A
B PRECEDES A

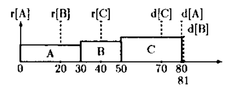


latest segment: A, C
lateness = e[A] - d[A] = e[C] - d[C] = 0
Z[A] = {A, B, C}
Z[C] = {C}
lowerbound = LB[A] = e[A] -d[A]
= LB[C] = e[C] - d[C] = 0
(globally optimal since lateness equal to
least lower bound of all open nodes)

Example 3.

r[A] = 0 r[B] = 20 r[C] = 40 **B EXCLUDES C**
c[A] = 30 c[B] = 20 c[C] = 30
d[A] = 80 d[B] = 81 d[C] = 70

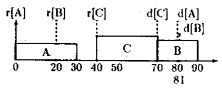
Root node of search tree:



latest segment: C
lateness = e[C] - d[C] = 80 - 70 = 10
Z[C] = {A, B, C}
lowerbound = LB2[C] = r'[C] + c[C] - d[C] = 40 + 30 - 70 = 0

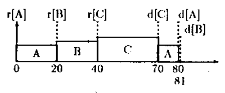
G1 = {B} G2 = {A}

C PRECEDES B



latest segment: B
lateness = e[B] - d[B] = 90 - 81 = 9
Z[B] = {C, B}
lowerbound = LB[B] = e[B] -d[B] = 90 - 81 = 9
(locally optimal since lateness equal to
lower bound - close this node)

C PREEMPTS A
B PREEMPTS A

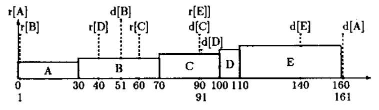


latest segment: A, C
lateness = e[A] - d[A] = e[C] - d[C] = 0
Z[A] = {A, B, C}
Z[C] = {C}
lowerbound = LB[A] = e[A] -d[A]
= LB[C] = e[C] - d[C] = 0
(globally optimal since lateness equal to
least lower bound of all open nodes)

Example 4.

r[A] = 0 r[B] = 1 r[C] = 60 r[D] = 40 r[E] = 90 **A EXCLUDES D C EXCLUDES E**
c[A] = 30 c[B] = 40 c[C] = 30 c[D] = 10 c[E] = 50 **A EXCLUDES B C EXCLUDES D**
d[A] = 161 d[B] = 51 d[C] = 90 d[D] = 91 d[E] = 140 **B EXCLUDES C D EXCLUDES E**
B PRECEDES D

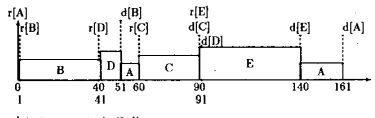
Root node of search tree:



latest segment: E
lateness = e[E] - d[E] = 160 - 140 = 20
Z[E] = {A, B, C, D, E}
lowerbound = LB2[E] = r'[E] + c[E] - d[E] = 90 + 50 - 140 = 0

G2 = {A} G1 = ∅

B PRECEDES A C PREEMPTS A
D PRECEDES A E PREEMPTS A

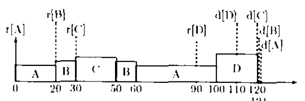


latest segment: A, C, E
lateness = e[A] - d[A] = e[C] - d[C] = e[E] - d[E] = 0
Z[A] = {B, D, C, E, A}
Z[C] = {C}
Z[E] = {E}
lowerbound = LB[A] = e[A] - d[A] = LB[C] = e[C] - d[C]
= LB[E] = e[E] - d[E] = 0
(globally optimal since lateness equal to
least lower bound of all open nodes)

Example 5.

r[A] = 0 r[B] = 20 r[C] = 30 r[D] = 90 **A EXCLUDES D**
c[A] = 60 c[B] = 20 c[C] = 20 c[D] = 20
d[A] = 122 d[B] = 121 d[C] = 120 d[D] = 110

Root node of search tree:

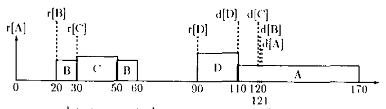


latest segment: D
lateness = e[D] - d[D] = 120 - 110 = 10
Z[D] = {A, B, C, D}
lowerbound = LB2[D] = r'[D] + c[D] - d[D] = 90 + 20 - 110 = 0

G1 = {A} G2 = {B, C}

(continued on next page)

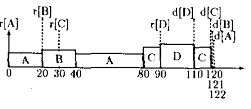
D PRECEDES A



latest segment: A
lateness = e[A] - d[A] = 170 - 122 = 48
Z[A] = {D, A}
lowerbound = LB[A] = e[A] -d[A] = 170 - 122 = 48
(locally optimal since lateness equal to
lower bound - close this node)

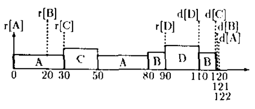
C ∈ G2 B ∈ G2

A PREEMPTS C
B PREEMPTS C
D PREEMPTS C



latest segment: C, D
lateness = e[C] - d[C] = e[D] - d[D] = 0
Z[C] = {B, A, D, C}
Z[D] = {D}
lowerbound = LB[C] = e[C] - d[C]
= LB[D] = e[D] - d[D] = 0
(locally optimal since lateness equal to
lower bound - close this node)

A PREEMPTS B
C PREEMPTS B
D PREEMPTS B



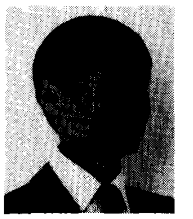
latest segment: B, D
lateness = e[B] - d[B] = e[D] - d[D] = 0
Z[B] = {C, A, D, B}
Z[D] = {D}
lowerbound = LB[B] = e[B] - d[B]
= LB[D] = e[D] - d[D] = 0
(globally optimal since lateness equal to
least lower bound of all open nodes.
Note that the schedule resulting from
C ∈ G2 is also globally optimal)

Acknowledgment

We are grateful to E. Margulis and K. C. Sevcik for helpful comments on earlier drafts. Helpful comments and suggestions from the anonymous referees are gratefully acknowledged.

References

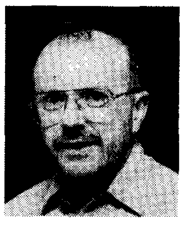
[1] J. Carlier, "Probleme a une machine," Institute de Programmation, Univ. Paris VI, manuscript, 1980.
[2] E. G. Coffman, Jr., *Computer and Jobshop Scheduling Theory.* New York: Wiley-Interscience, 1976.
[3] S. R. Faulk and D. L. Parnas, "On synchronization in hard-real-time systems," *Commun. ACM*, vol. 31, Mar. 1988.
[4] M. R. Garey and D. S. Johnson, "Scheduling tasks with non-uniform deadlines on two-processors," *J. ACM*, vol. 23, July 1976.
[5] ——, "Two-processor scheduling with start-times and deadlines," *SIAM J. Comput.*, vol. 6, Sept. 1977.
[6] ——, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* San Francisco, CA: Freeman, 1979.
[7] M. R. Garey, D. S. Johnson, B. B. Simons, and R. E. Tarjan, "Scheduling unit-time tasks with arbitrary release times and deadlines," *SIAM J. Comput.*, vol. 10, May 1981.
[8] M. J. Gonzalez, Jr., "Deterministic processor scheduling," *Comput. Surveys*, vol. 9, Sept. 1977.
[9] D. Gunsfield, "Bounds for naive multiple machine scheduling with release times and deadlines," *J. Algorithms*, vol. 5, 1984.
[10] E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan, "Recent developments in deterministic sequencing and scheduling: A survey," in *Proc. NATO Advanced Study and Research Institute on Theoretical Approaches to Scheduling Problems*, Durham, England, July 1981; also in *Deterministic and Stochastic Scheduling*, M. A. H. Dempster *et al.*, Eds. Dordrecht, The Netherlands: D. Reidel.
[11] C. L. Lui and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, Jan. 1973.
[12] G. K. Manacher, "Production and stabilization of real-time task schedules," *J. ACM*, vol. 14, July 1967.
[13] C. Martel, "Preemptive scheduling with release times, deadlines, and due dates," *J. ACM*, vol. 29, July 1982.
[14] G. McMahon and M. Florian, "On scheduling with ready time and due dates to minimize maximum lateness," *Oper. Res.*, vol. 23, 1975.
[15] A. K. Mok and M. L. Detouzos, "Multiprocessor scheduling in a hard real-time environment," in *Proc. 7th IEEE Texas Conf. Computing Systems*, Nov. 1978.
[16] A. K. Mok, "Fundamental design problems of distributed systems for the hard-real-time environment," Ph.D. dissertation, Dept. Elec. Eng. Comput. Sci., Massachusetts Inst. Technol., Cambridge, MA, May 1983.
[17] ——, "The design of real-time programming systems based on process models," in *Proc. IEEE Real-Time Systems Symp.*, Dec. 1984.
[18] B. Simons, "Multiprocessor scheduling of unit-time jobs with arbitrary release times and deadlines," *SIAM J. Comput.*, vol. 12, May 1983.
[19] W. Zhou, K. Ramamrithan, and J. Stankovic, "Preemptive scheduling under time and resource constraints," *IEEE Trans. Comput.*, Aug. 1987.



Jia Xu received the Docteur en Sciences Appliquées degree in computer science from the Université Catholique de Louvain, Belgium, in 1984.

He is presently an Assistant Professor in the Department of Computer Science at York University, North York, Ont., Canada. From 1984 to 1985 he was a postdoctoral fellow at the University of Victoria, Victoria, B.C., Canada. From 1985 to 1986 he was a postdoctoral fellow at the University of Toronto, Toronto, Ont., Canada. His research interests include real-time systems, scheduling, and database management systems.

Dr. Xu is a member of the Association for Computing Machinery.



David Lorge Parnas, born February 10, 1941, is a Professor at Queen's University in Kingston, Ont., Canada, where he is a project leader and principal investigator for the Telecommunications Research Institute of Ontario. He is interested in all aspects of computer systems engineering. His special interests include program organization, program semantics, precise computer system documentation, process structure, process synchronization, and precise abstract specifications. He initiated and led an experimental redesign of a hard-real-time system, the on-board flight program for the U.S. Navy's A-7 aircraft, in order to evaluate a number of software engineering principles. More recently he has advised the Atomic Energy Control Board on the use of safety-critical real-time software in a new nuclear plant. Previously, he was Lansdowne Professor of Computer Science at the University of Victoria, Victoria, B.C., Canada. He was also Principle Investigator of the Software Cost Reduction Project at the Naval Research Laboratory in Washington, DC. He has also taught at Carnegie-Mellon University, the University of Maryland, the Technische Hochschule Darmstadt, and the University of North Carolina at Chapel Hill.

Dr. Parnas was the first winner of the "Norbert Wiener Award for Professional and Social Responsibility," given annually by Computing Professionals for Social Responsibility, and has an honorary doctorate from the ETH in Zurich.