Handling Duration Uncertainty in Meta-Level Control of Progressive Processing

Abdel-Illah Mouaddib*
CRIL-IUT de Lens-Université d'Artois
Rue de l'université, S. P. 16
62307 Lens Cedex France
mouaddib@cril.univ-artois.fr

Shlomo Zilberstein†
Computer Science Department
University of Massachusetts
Amherst, MA 01003 U.S.A.
shlomo@cs.umass.edu

Abstract

Progressive processing is a resource-bounded reasoning technique that allows a system to incrementally construct a solution to a problem using a hierarchy of processing levels. This paper focuses on the problem of meta-level control of progressive processing in domains characterized by rapid change and high level of duration uncertainty. We show that progressive processing facilitates efficient run-time monitoring and meta-level control. Our solution is based on an incremental scheduler that can handle duration uncertainty by dynamically revising the schedule during execution time based on run-time information. We also show that a probabilistic representation of duration uncertainty reduces the frequency of schedule revisions and thus improves the performance of the system. Finally, an experimental evaluation shows the contributions of this approach and its suitability for a data transmission application.

1 Introduction

Progressive processing [Mouaddib, 1993] is a problem-solving technique that allows a system to trade-off solution quality against computational resources. This technique is suitable in situations where it is not feasible (computationally) or desirable (economically) to compute the best result. Progressive processing shares the motivation of such resource-bounded reasoning techniques as *flexible computation* [Horvitz, 1987], *anytime algorithms* [Boddy and Dean, 1994; Zilberstein, 1995; 1996], and *design-to-time* [Garvey and Lesser, 1993]. The distinctive characteristic of progressive processing is the use of a multi-level deliberation hierarchy in order to gradually transform an approximate solution into a precise one. The mapping from the set of inputs (problem instances) to the set of outputs (solutions) is based on progressive utilization of data and knowledge. This incremental process is facilitated by using a hierarchical structure of input elements defined by the system's designer [Mouaddib and Zilberstein, 1995]. This mapping is especially suitable for domains where the reasoner uses abstraction to structure the search space (as in hierarchical planning), and for problems that require the result to be expressed at varying levels of detail (as in model-based diagnosis).

In systems based on this technique, each time-constrained component is designed in such a way that it can produce a usable approximate solution within the available run-time. However the use of progressive processing as a component of a real-time system introduces a problem of execution monitoring: determining how long to run. This kind of scheduling problems are common in real-time AI applications, such as medical monitoring [Hayes-Roth, 1990], real-time data transmission [Millan-Lopez *et al.*, 1994], speech processing [Feng and Liu, 1993], mobile robot navigation [Zilberstein, 1996] and flexible manufacturing. The real-time data transmission application that we address in this paper is another example. These application require a meta-level control mechanism that can deal with (1) dynamic environment, (2) limited amount of time to produce a response, and (3) uncertainty regarding the duration of problem-solving.

The focus of this work is handling duration uncertainty. Most of the existing approaches to the problem do not deal with duration uncertainty and their schedulers work in isolation; the schedule runs for the predetermined length of time regardless of the situation [Boddy and Dean, 1994; Liu *et al.*, 1991; Millan-Lopez *et al.*, 1994]. Hansen and Zilberstein [Hansen and Zilberstein, 1996] show that monitoring the progress of problem solving allows a system to take advantage of the information gathered during execution time and to improve the overall performance. Garvey and Lesser [Garvey and Lesser, 1993] introduce a *design-to-time* technique that represents duration uncertainty and revises the schedule during execution. We address two limitations of that approach: (1) its time interval repre-

*Support for this author is provided in part by the Ganymède II project of the contract Plan Etat/Nord-Pas-De-Calais and by MENESR.

†Support for this author is provided by the National Science Foundation under grants IRI-9624992, IRI-9634938, INT-9612092, and by the U.S. Air Force under grant F30602-95-1-0012.

sentation of duration uncertainty is not sufficiently informative when the duration variation is large; and (2) its scheduler is assumed to always complete its processing. This design-to-time approach is suitable in domains characterized by low level of uncertainty so that the scheduler is significantly faster than the evolution of the controlled process. We suggest that it is beneficial to design the scheduler itself as an incremental process so that it can handle rapid change in the environment and unexpected interruption before completing its execution.

This paper focuses on the problem of meta-level control of computational resources in domains characterized by rapid change and duration uncertainty. We show that the task structure of progressive processing facilitates efficient run-time monitoring and meta-level control. The approach we present is based on an interruptible algorithm which incrementally builds a schedule and returns an approximate complete schedule when it is interrupted unexpectedly. This approach can handle rapid change and a large deviation from the predicted schedule. The incremental scheduler that we present allows: (1) to initially construct schedule that contains the first reasoning level for each task and to then refine it progressively by introducing additional reasoning levels for each task, (2) to revise the schedule when a significant deviation is detected at run-time, and (3) to reduce the frequency of revision using information on duration uncertainty.

Section 2 presents a data transmission application for which our approach was implemented. Section 3 presents a formal definition of the problem and the initial task structure. A description of our approach based on an incremental scheduling and a revision technique is given in Section 4. Section 5 presents an empirical comparison of our approach with an adaptation of the design-to-time approach to handle progressive processing. We conclude with a summary of the contributions and future work.

2 Real-Time Data Transmission Application

We examine a data transmission application that provides real-time communication services. These services include time constraints on the duration of transmission and a deadline for data delivery. The duration of transmission is the time interval between the point at which the data is generated and the point at which it is delivered. This duration is uncertain because of the variation on the behavior of the communication network. Data misses its deadline whenever the duration of transmission exceeds the maximum permitted time.

Our approach is designed to manage real-time World-Wide Web (WWW) services providing information on staff members of our laboratory. This information may include textual data, video frames and voice. The system must satisfy asynchronous requests of information by taking their deadlines into account. The implementation consists of eleven (11) WWW pages, each of which contains information on one member of the laboratory. Each page consists of textual data, video frames and voice. Different requests for information may refer to the same page. The information system must handle (1) asynchronous requests (dynamic and rapid change aspect of the application); and (2) uncertainty regarding the behavior of the network (uncertainty on execution time). To apply the progressive processing technique, we assume that each WWW page could be transmitted as three separate packages of information (text, video frames and voice). The incremental process of satisfying a request is based on dividing it into a sequence of information packages. Consequently, the requests could be satisfied at varying levels of detail. Our experimental evaluation is based on a simulation of arrival of information requests and transmission of information packages.

3 Formal Framework

3.1 Description of the Problem

Our framework relaxes two assumptions that are common in classical scheduling systems. First, they optimize performance by satisfying the most important requests and if time permits they satisfy additional requests. These approaches neglect the fact that each request could be satisfied at varying levels of detail. It has been demonstrated that requests can be logically decomposed into two parts: a mandatory part and an optional part [Liu *et al.*, 1991]. This structure facilitates the development of more flexible systems that allow to satisfy the mandatory parts of all the requests and then improving performance by satisfying some of the optional parts. This approach is applies to problems that could be solved at varying levels of detail, such as hierarchical planning [Knoblock, 1994] and incremental diagnosis [Hayes-Roth, 1990]. Second, classical scheduling systems ignore the deviation of execution time from the predetermined *expected* length. In fact, data transmission applications are characterized by a high level of uncertainty regarding transmission time that cannot be ignored.

More precisely, the problem we solve consists of a set $P = \{P_1, \ldots, P_n\}$ of individual problems such that:
- P is constructed dynamically; a new problem is added to the set when a new request arrives,
- each problem P_i has a deadline D_i to respect,
- each problem P_i could be solved at varying levels of detail using a hierarchy of processing levels, and
- each processing level has probabilistic information characterizing its duration and duration uncertainty.

3.2 Progressive Processing

A progressive processing unit is a composite unit α of processing levels L_α^i ordered by a linear precedence-constraint graph. The level L_α^i can begin execution only after the level L_α^{i-1} completes. The level L_α^i is the immediate successor of L_α^{i-1}, and the output of L_α^{i-1} is one of the inputs of L_α^i. The output of the last level

$L_\alpha^{n_\alpha}$ is the (best) output of the unit α. In general, the output of a unit can be used by any successive unit as soon as the first level has finished its processing. We assume that the information necessary for successive units is produced by the first level. The hierarchical structure of each unit allows to characterize precisely the tradeoff between computation time and the quality of the result. The solution quality is discretizd into a finite number of levels, $q_{L_\alpha^1}$, $q_{L_\alpha^2}$,..., $q_{L_\alpha^{n_\alpha}}$, where $q_{L_\alpha^1}$ is the lowest quality and $q_{L_\alpha^{n_\alpha}}$ is the highest quality returned respectively by the first level L_α^1 and the last level $L_\alpha^{n_\alpha}$. The purpose of monitoring is to optimize the time/quality tradeoff offered by each unit.

3.3 Formal Framework

Scheduling progressive processing units consists of deciding how long to run each unit taking its progressive structure into account and satisfying the timing constraints imposed on the problem. Before elucidating this problem and its solution, we briefly present some general notation and concept definitions used by the meta-level control algorithm.

Stating the problem We consider a set $\mathcal{A}$ comprising n progressive processing units (PRUs) α, β,..., γ sorted according to the deadlines (*earliest-deadline-first*) of the corresponding problems P_i in P that they solve. The problem is to find for each PRU the optimal subset of its set of processing levels such that all deadlines are respected. This decision is made under uncertainty regarding the execution time of each processing level and regarding future changes of the set $\mathcal{A}$.

Progressive processing unit Each PRU α is represented by a linear graph $L_\alpha^1 \rightarrow L_\alpha^2 \rightarrow \ldots \rightarrow L_\alpha^{n_\alpha}$, denoted LG_α=($\mathcal{L}_\alpha$, Succ_Level) where:
• $\mathcal{L}_\alpha = \{L_\alpha^i \mid 1 \leq i \leq n_\alpha\} \bigcup \{\epsilon\}$, ϵ means no successor
• $Succ\_Level: \mathcal{L}_\alpha \rightarrow \mathcal{L}_\alpha$,

$$Succ\_Level(L_\alpha^i) = \begin{cases} L_\alpha^{i+1} & \text{if } i < n_\alpha \\ \epsilon & \text{if } i = n_\alpha \end{cases} \quad (1)$$

Processing level formulation Each level L_α^i is characterized by the triplet (c_α^i, v_α^i, V($q_{L_\alpha^i}$)) where:
• c_α^i, v_α^i: its computation time average and variance.
• V($q_{L_\alpha^i}$): its intrinsic value of solution quality.
The average and the variance are dynamically determined and updated by statistical analysis on the behavior of this level. The average represents the most likely computation time. The variance is used for judging whether the deviation during execution from the predicted schedule is important.

Utility of a processing level The utility U_α^i of a reasoning level L_α^i is defined as follows:

$$U_\alpha^i = V(q_{L_\alpha^i}) - Cost(c_\alpha^i) \quad (2)$$

Where $Cost(c_\alpha^i)$ is an application-dependent cost function provided by the user. This utility is used to attach a comprehensive value to each processing level at any given time. Based on this value, a utility-based approach is used in order to select processing levels to be included in the schedule. Similarly, at execution time, the utility of each processing level is used to determine which level is to be added/removed from the current schedule when execution is faster/slower than originally predicted.

Framework used We represent the ordered set $\mathcal{A}$ by a graph ($\mathcal{G}$, Succ_Unit, Succ_Node) (Figure 1) where:
• $\mathcal{G} = \{L_\alpha^i \in LG_\alpha | \forall\ \alpha\ \in\ \mathcal{A}\}$.
• $Succ\_Unit: \mathcal{A} \rightarrow \mathcal{A}$,
Succ_Unit(α) = β, β is the immediate successor in $\mathcal{A}$,
• $Succ\_Node: \mathcal{G} \rightarrow \mathcal{G}$,

$$Succ\_Node(L_\alpha^i) = \begin{cases} Succ\_Level(L_\alpha^i) & \text{if } i < n_\alpha \\ L^1_{Succ\_Unit(\alpha)} & \text{if } i = n_\alpha \end{cases} \quad (3)$$

This structure allows to perform utility-based schedul-

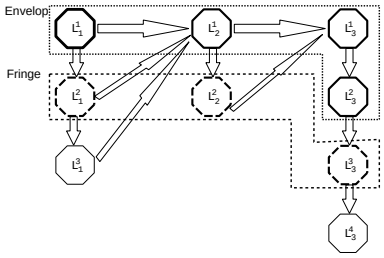

Figure 1: Example showing the framework

ing by incrementally inserting nodes (processing levels) into the current schedule. We now define several concepts necessary to describe the scheduling process.
• The *envelop*, $\mathcal{E}$, is a subgraph of $\mathcal{G}$ containing all the levels scheduled for execution (nodes drawn with bold lines in Figure 1). This is similar to the notion of a *closed list* in search algorithms. The envelop is a dynamic structure that can be revised during execution.
• The *fringe*, $\mathcal{F}$, is the set of direct successors to the nodes in $\mathcal{E}$ (nodes drawn with dotted lines in Figure 1). This is similar to the notion of an *open list* in search that contains nodes on the frontier of the search tree that are candidates for expansion.
• The *frozen space*, $\mathcal{Z}$, is a set of branches pruned during the scheduling cycle. It contains the levels that violate some deadlines during the scheduling phase. We save these levels because during the revision of the schedule, these levels may be added to the search space and eventually they may be added to the schedule.

4 Incremental Scheduling

The main design goal of the incremental scheduler is the ability to return an approximate schedule for all the PRUs of $\mathcal{A}$ at any time and to revise this schedule when

a deviation from the predetermined schedule is detected at run-time. These two properties allow the system to deal with domains characterized with rapid change and a high level of uncertainty.

4.1 Conceptual Description

With the PRU structure and formal framework described above, scheduling could be seen as the problem of finding an optimal path that visits the maximum number of levels in the graph $\mathcal{G}$ without violating any deadline. There are different possible paths with different qualities. Our strategy consists of finding a minimal schedule that includes all the PRUs and refining it progressively. The minimal schedule has the lowest quality since it visits only the first node of each LG (in the example shown in Figure 1, the minimal envelop is $\{L_1^1, L_2^1, L_3^1\}$). The path with the highest quality is the path that visits all the nodes (in the example, the maximal envelop is $\{L_1^1, L_1^2, L_1^3, L_2^1, L_2^2, L_3^1, L_3^2, L_3^3, L_3^4\}$). The scheduler starts its processing by building the schedule with the lowest quality (the minimal envelop) and refining it by inserting additional nodes into the graph as long as all the deadlines are respected. The incremental processing of the scheduler is guided by the progressive structure of the PRUs and by the the (easy to construct) *fringe*.

4.2 Utility-Based Scheduling Algorithm

The construction of the schedule is based on a series of cycles of expansion of the current envelop. This expansion consists of inserting levels of the fringe into the envelop. This process is repeated until a maximal envelop is reached (i.e., any further expansion leads to a violation of a deadline) or until an external even causes the interruption of scheduling. At each cycle, a schedule is available and its quality is improved from one cycle to another. The algorithm consists of the following steps:

• **Initialization step**:

$$\mathcal{E} = \mathcal{Z} = \emptyset \; and \; \mathcal{F} = \{L_\alpha^1 \in \mathcal{G} \mid \forall \alpha \in \mathcal{A}\} \quad (4)$$

• **Expansion step**:

This step consists of extending $\mathcal{E}$ to all the levels in $\mathcal{F}$:

$$\mathcal{E} = \mathcal{E} \biguplus \mathcal{F} \; and \; \mathcal{F} = \emptyset \quad (5)$$

The operator $\biguplus$ allows to insert levels while respecting the structure of the graph $\mathcal{G}$ and thus at each cycle, $\mathcal{E}$ is a subgraph of $\mathcal{G}$ (as shown in Figure 1).

• **Test of feasibility step**:

The schedule fails when one deadline is violated:

$$\exists \gamma \in \mathcal{A}: \sum_{\{\delta \in \mathcal{A},\ D_\delta \leq D_\gamma\}} \sum_{i=1}^{i=d_\delta} c_\delta^i > D_\gamma, \quad (6)$$

$$\text{where } \{L_\delta^1, \ldots, L_\delta^{d_\delta}\} \subset \mathcal{E}$$

If the schedule fails, go to the *approximation step*, otherwise go to the *new cycle step*.

• **Approximation step**:

The level with the lowest utility, noted L_{min}, when is discarded when the schedule fails. The level L_{min} is selected among the levels L_γ^k inserted by the last expansion cycle.

$$L_{min} = arg(MIN_{L_\gamma^k}(U_{L_\gamma^k})) \quad (7)$$

The branch containing the successors nodes in LG_ψ is pruned from $\mathcal{G}$ and is placed in $\mathcal{Z}$. Formally:

$$\mathcal{G} = \mathcal{G} \ominus \{Succ\_Level(L_{min}), \ldots, L_\psi^{n_\psi}\} \quad (8)$$

$$\mathcal{Z} = \mathcal{Z} \bigcup \{Succ\_Level(L_{min}), \ldots, L_\psi^{n_\psi}\} \quad (9)$$

The operator $\ominus$ performs the following operation: Succ_Node(L_{min})=ϵ. Afterwards, go to the *test of feasibility step*.

• **New fringe step**:

For each processing level L_α^i inserted in $\mathcal{E}$ by the last expansion cycle, we insert into the fringe its successor (if it exists). Formally:

$$\mathcal{F} = \{Succ\_Level(L_\alpha^i) \neq \epsilon \mid L_\alpha^i \in \mathcal{E}\} \quad (10)$$

If the fringe is not empty go to the *expansion step*, otherwise stop the algorithm and return the envelop $\mathcal{E}$.

4.3 Utility-Based Revision Algorithm

This algorithm adjusts the *schedule* by taking into account the actual progress made at run-time. After the execution of each PRU, the algorithm examines the actual execution time and determines whether an important deviation has occurred. A deviation is important when it exceeds the variance v_α^i of one processing level which is not yet scheduled or executed. This algorithm allows to optimally revise the schedule and to update statistical information concerning the executed level. The processing time gained/lost, Dev, is used to insert/remove levels and correspondingly increase/decrease the global performance efficiently. The algorithm inserts the level L_α^i of $\mathcal{F}$ or $\mathcal{Z}$ with the maximum utility U_α^i and removes the level L_α^i of $\mathcal{E}$ with the minimum U_α^i. If there is more than one PRU having a maximum/minimum utility we choose the most urgent PRU when we want to insert a level and the latest PRU when we want to remove a level. The revision algorithm is based on the following rules:

• If Dev is positive, this means that the amount of time was lost during execution (i.e., execution is slower than predicted): (1) select the level such as $L_{min} = arg(MIN_{L \in \mathcal{E}}(U_L))$, (2) use the rules of the *approximation step* of the scheduling algorithm.

• If Dev is negative, this means that the amount of time was gained during execution (i.e., execution is faster than predicted): (1) select the level such as $L_{max} = arg(MAX_{L \in (\mathcal{F} \bigcup \mathcal{Z})}(U_L))$, (2) add this level to $\mathcal{E}$, (3) if the selected level is in $\mathcal{F}$, then its corresponding branch $\{Succ\_Level(L_{max}), \ldots, L_\psi^{n_\psi}\}$ is reintegrated into $\mathcal{G}$:

$$\mathcal{G} = \mathcal{G} \biguplus \{Succ\_Level(L_{max}), \ldots, L_\psi^{n_\psi}\} \quad (11)$$

4.4 Properties

To summarize, here are the main properties of our approach:

• Its scheduler could be interrupted at any time and it returns an approximate global schedule (global means that all the PRUs are included). As a result, this approach can deal with applications characterized by rapid change and a high level of uncertainty.

• The probabilistic representation of duration uncertainty reduces the deviation from the predetermined schedule during execution. It also helps reducing the frequency of revising the schedule and thus limits the effect of revision time on the performance of the system.

5 Experimental Evaluation

To evaluate our scheduling and monitoring technique, we compare its performance to the design-to-time technique. The comparison examines two fundamental questions: (1) the ability of each approach to handle duration uncertainty that is typical in such applications as real-time data transmission; and (2) the advantage of using a resource-bounded scheduler in domains characterized by rapid change and high level of uncertainty. Before assessing both approaches, let us give a brief overview of design-to-time. This approach assumes the existence of multiple methods for each subtask with each method having different duration and quality. The problem is to design a solution to a problem that uses all the available time to maximize solution quality. design-to-time tolerates uncertainty in its prediction if monitoring can be performed quickly and each method performs as expected or within a small variation. The experimental results confirm these characteristics.

5.1 Experiment Design

We describe first how we have transformed the PRU structure to multiple methods used by design-to-time. The linear precedence-constraint graph of each PRU is mapped to a set of methods designed as follows: the first method M_1 consists of the first level (the first node in LG) while the i^{th} method M_i consists of the subgraph of the LG containing the first i^{th} levels [Mouaddib, 1996]. With this adaptation, we can evaluate both approaches for different problem instances generated based on a given set of size and time parameters.

Since the WWW data transmission application is under development right now, we tested the scheduler with synthetic data simulating this application. The initial problem is specified in a *PRU-language* allowing us to create PRUs for the specific application. We have collected data on the scheduling and the execution phase for both the incremental scheduler and the design-to-time technique. The data for each problem instance includes *intrinsic-utility, total cpu consumed* and the *frequency of revision.* Finally, we compare the results based on the notion of *global utility*, $\mathcal{GU}$, and the notion of revision frequency, $Freq$. The global-utility is computed as follows:

$$\mathcal{GU} = (\sum_{\forall\ L\ executed} U_\alpha^L)\ -\ Cost(total\_cpu) \quad (12)$$

Where U_α^i is the intrinsic utility of an executed processing level. We compare $Freq$ and $\mathcal{GU}$ for different problem sizes and time.

Handling duration uncertainty

The frequency of revision reflects the degree to which an approach is suitable for handling duration uncertainty. We measured the frequency and global utility as a function of the size of the application. Problem instances were generated with execution time variation of 20% (this figure is quite conservative for data transmission applications). The table (Figure 2) summarizes the frequency of revision of our approach (IS) and design-to-time (DTT): Figure 3 shows the evolution of the global utility over size in both approaches.

Size	1	2	3	4	5	6	7	8	9	10
$Freq$ in IS	0	1	0	0	0	0	2	1	1	1
$Freq$ in DTT	0	1	1	1	1	2	3	2	4	6

Figure 2: Revision frequency according to size

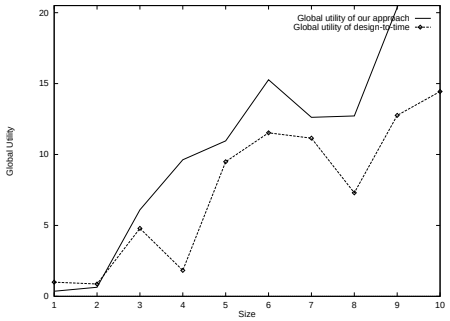

Figure 3: Global utility according to size

Resource-bounded scheduling

This experiment measures the performance of both approaches when the scheduling algorithm operates under time constraints (Figure 4). The experiment tests the degree to which an approach is suitable for dynamic environments in which the scheduler may not have enough time to complete its processing. For this experiment, we use a size of 10 and a variable allocation of time to the scheduler of both approaches. We measured the intrinsic utility that allows to compute $\mathcal{GU}$. Design-to-time requires to finish its processing before delivering a complete schedule for all tasks. In this experiment, we measure the intrinsic utility of the current available schedule that is not complete for all tasks but only for the earlier ones.

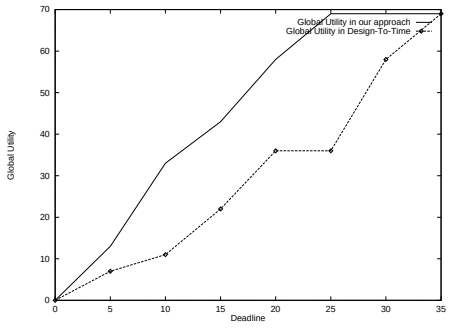

Figure 4: Global utility according to time

5.2 Summary of results

• **Handling duration uncertainty:** our approach is more efficient than design-to-time in handling duration uncertainty. The main reason is the time interval representation of duration uncertainty used by design-to-time which is not sufficiently informative and leads to more schedule revisions. The higher frequency of revision has a negative effect on the overall performance of the approach as shown in the Figure 3.

• **Resource-bounded scheduling:** design-to-time schedules tasks one by one by selecting for each task the appropriate method to optimally trade solution quality against computational resources. If the scheduling is interrupted before completing its processing, the latest tasks are not scheduled. This situation arises in applications that must respond to frequent and rapid state change. In contrast, our incremental scheduler, with its incremental processing, is able to be interrupted at any time and to return a solution. We show in Figure 4 that our approach is more suitable to these applications than design-to-time.

6 Conclusion and Future work

We describe in this paper an approach to scheduling progressive processing units in domains characterized by rapid change and duration uncertainty such as in data transmission applications. Our approach is based on a utility-directed greedy scheduling algorithm that produces a minimal schedule of all the tasks and refines it when time permits. This is a local optimization approach to a scheduling problem that is hard to optimize globally. The incremental structure of the scheduler, its interruptibility, and its ability to dynamically revise the schedule during execution time make it an attractive resource-bounded scheduling technique. Experimental evaluation shows that for the type of progressive processing tasks that we are interested in, the incremental scheduler has advantages over the design-to-time technique. Future work is concerned with alternative representations and propagation of duration uncertainty by mapping the graph $\mathcal{G}$ to a Bayesian network [Mouaddib, 1996]. Another future direction is concerned with applying this model to non-linear structure of PRUs.

References

[Boddy and Dean, 1994] M. Boddy and T. Dean. Deliberation scheduling for problem solving in time-constrained environments. *Artificial Intelligence*, 67:245–285, 1994.

[Feng and Liu, 1993] W. Feng and J.W.S Liu. An extended imprecise computation model for time-constrained speech processing and generation. In *IEEE Workshop on Real-Time Applications*, pages 76–80, 1993.

[Garvey and Lesser, 1993] A. Garvey and V. Lesser. Design-to-time real-time scheduling. *IEEE Transactions on systems, Man, and Cybernetics*, 23(6), 1993.

[Hansen and Zilberstein, 1996] Hansen and Zilberstein. Monitoring the progress of anytime problem-solving. *AAAI-96*, 1996.

[Hayes-Roth, 1990] B. Hayes-Roth. Architectural foundation for real-time performance in intelligent agents. *Journal of Real-Time Systems*, 2(1), 1990.

[Horvitz, 1987] E.J. Horvitz. Reasoning about beliefs and actions under computational resource constraints. In *Workshop UAI-87*, 1987.

[Knoblock, 1994] C.A. Knoblock. Autmatically generating abstractions for planning. *Artificial Intelligence*, 68:243–302, 1994.

[Liu *et al.*, 1991] J. Liu, K. Lin, W. Shih, A. Yu, J. Chung, and W. Zao. Algorithms for scheduling imprecise computations. *IEEE Transactions on Computer*, 24(5):58–68, May 1991.

[Millan-Lopez *et al.*, 1994] V. Millan-Lopez, W. Feng, and J.W.S Liu. Using the imprecise-computation technique for congestion control on a real-time traffic switching element. In *International Conference on Parallel and Distributed Systems*, 1994.

[Mouaddib and Zilberstein, 1995] A.-I Mouaddib and S. Zilberstein. Knowledge-based anytime computation. In *IJCAI*, pages 775–781, 1995.

[Mouaddib, 1993] A.-I Mouaddib. Contribution au raisonnement progressif et temps réel dans un univers multi-agents. *PhD, University of Nancy I, (in French)*, 1993.

[Mouaddib, 1996] A.-I Mouaddib. Progressive reasoning in intelligent systems. In *AAAI Fall Symposium on Flexible Computation, Research Summary Report*, pages 183–185, 1996.

[Zilberstein, 1995] S. Zilberstein. Optimizing decision quality with contract algorithms. In *IJCAI-95*, pages 1576–1582, 1995.

[Zilberstein, 1996] S. Zilberstein. Using anytime algorithms in intelligent systems. *AI Magazine*, 17(3):73–83, 1996.