Design-to-Criteria Scheduling for Intermittent Processing [*]

Thomas A. Wagner and Victor Lesser
Computer Science Department
Lederle Graduate Research Center
University of Massachusetts
Amherst, MA 01003-4601
Email: {wagner,lesser}@cs.umass.edu

UMass Computer Science Technical Report 1996-81

November 16, 1996

Revised October 22, 1999

Abstract

TÆMS is an abstract task modeling framework used to describe the major components of tasks and to reason about trade-offs between various possible approaches of performing tasks. To better model certain classes of tasks that arise in distributed environments, we have enhanced the model to more accurately portray two classes of intermittent processing activities – those that contain *embedded* delays, such as embedded I/O, and those that are affected by *external* delays, e.g., slow propagation of results from a different location. The enhancements are important because they help identify situations where local processing resources are under utilized and possibly idle. Accordingly, we must modify the Design-to-Criteria scheduling process and leverage the enhanced model to produce more efficient schedules. Modifications include determining when it is possible to overlap actions that contain embedded delays, and the ramifications to their execution profiles, and deciding what actions may be performed during an external delay.

[*]This material is based upon work supported by the National Science Foundation under Grant No. IRI-9523419, the Department of the Navy, Office of the Chief of Naval Research, under Grant No. N00014-95-1-1198, and via a subcontract from Boeing Helicopter which is being supported by DARPA under the RaDEO program (contract number 70NANB6H0074). The content of the information does not necessarily reflect the position or the policy of the Government, National Science Foundation, or Boeing Helicopter and no official endorsement should be inferred.

1 Introduction

TÆMS (Task Analysis, Environment Modeling, and Simulation) [3, 4, 5] is a task modeling framework used to describe and reason about complex problem solving processes. TÆMS models are abstractions of problem solving processes; they represent major tasks and major decision points, interactions between tasks, and resource constraints but they do not describe the intimate details of each primitive action. Alternative approaches for performing tasks are represented explicitly in TÆMS and all primitive actions are statistically characterized in three dimensions: quality, cost and duration. The explicit representation of alternative approaches to performing tasks, and the statistical characterization of primitive actions, allow TÆMS users and tools to reason about the quality, cost, and duration trade-offs of different possible courses of action. TÆMS models are the grounding element and medium of exchange for Design-to-Time [13, 15] and Design-to-Criteria scheduling [20], and multi-agent [7, 6, 8, 2] coordination research, and are being used in Cooperative-Information-Gathering [18, 19, 10], collaborative distributed design [9], and distributed situation assessment [1] projects.

Over the last few years TÆMS has been extended to increase its representational power for modeling "real world" problem solving systems. Most notable among these extensions are the additions of explicit representations of uncertainty [14] and multiple outcomes [16]. The uncertainty enhancement entails representing and reasoning about actions and trade-offs using discrete probability distributions rather than simple expected values. The detail provided by the full uncertainty representation enables probabilistic reasoning about different aspects of each dimension. For example, using the uncertainty representation tools can consider the probability that a particular range of values will result, or measure the degree of uncertainty about particular values or ranges of values, rather than being limited to the expected value alone. The outcomes enhancement associates multiple possible outcomes with each primitive action, each with its own probability distribution and its own set of outgoing task interactions. This allows TÆMS users to more accurately model the semantics of real-world actions where different results may have very different effects (interactions) on the problem solving process. Similarly, different results may have very different statistical descriptions and separating the outcomes facilitates reasoning from an outcome perspective, rather than an aggregate perspective, where desired.

In this work, we have enhanced the model to more accurately portray two classes of intermittent processing tasks – those that contain embedded delays, such as embedded I/O, and those that are affected by external delays, e.g., slow propagation of computational results from elsewhere on the network. Modeling these two classes of intermittent processing in TÆMS is only half the battle. To leverage the representation and improve *processor*[1] utilization we must also address the enhanced model from the scheduling perspective. This entails knowing when certain actions must be delayed pending result propagation and detecting when it is possible to perform multiple actions within a single time interval. In the sections that follow we provide a brief introduction to TÆMS and Design-to-Criteria scheduling, define the model enhancements, and describe how the scheduling process leverages the more expressive representation to build better schedules.

2 TÆMS Overview

Graphically the model is a tree where interior nodes, called *tasks*, denote abstract high-level problem solving activities and leaves, *methods*, represent executable actions. In TÆMS actions are modeled statistically along three dimensions, quality, cost, and time. Quality is a deliberately ab-

[1]We will use the term *processor* to describe a generic execution unit, e.g., a cpu or an agent.

stract domain-independent concept that describes the "goodness" of performing a particular action. Thus, different applications have different notions of what corresponds to model quality. Duration describes the amount of time that a method will take to execute. Cost describes the financial or opportunity cost inherent in performing the action modeled by the method. The statistical characteristics of the three dimensions are described via discrete probability distributions associated with each method.

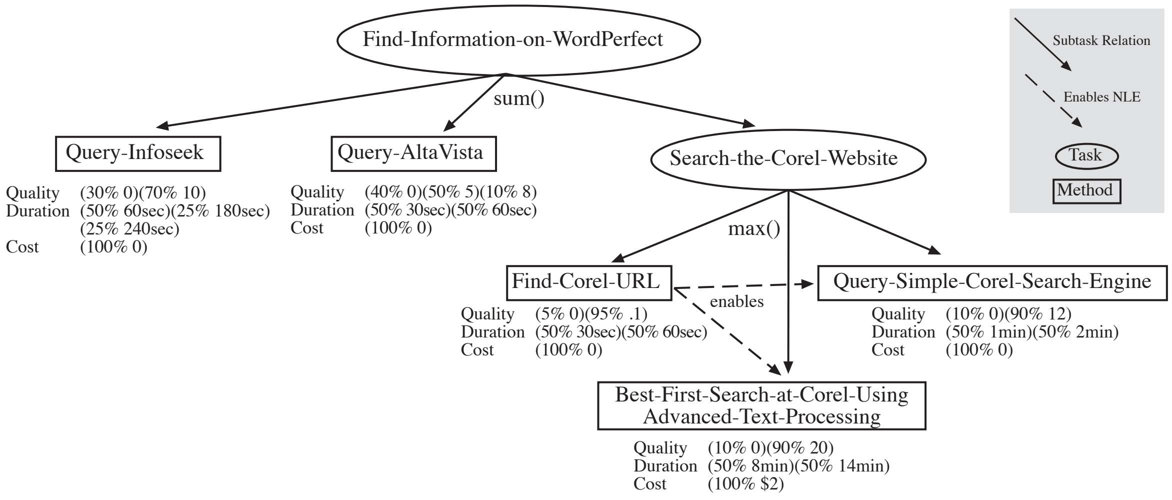
TÆMS diverges from traditional hierarchical representations in that different alternatives for achieving a task are expressed explicitly and reasoning about trade-offs is a first class activity. For example, a task may have multiple submethods, one or more of which may be executed to achieve the task. Different combinations of submethods have different cost, quality, and duration characteristics and different degrees of uncertainty about these characteristics. The objective in any TÆMS related activity is to achieve quality for the task structure root, or *task group*, which is synonymous with achieving the high-level task.[2] As with most hierarchical representations the high-level task is achieved by achieving some combination of its subtasks – quality achievement is a ubiquitous goal. High-level TÆMS tasks accumulate quality from their subtasks, which get quality from their subtasks recursively until the methods are reached, according to *quality accumulation functions* (qaf). Qafs are approximations that model how utilities are calculated and propagated in the problem solving process described by the model. The primary TÆMS qafs are `max()`, which is somewhat analogous to a logical OR, `min()`, comparable to logical AND, and `sum()` where any member of the power set [3] of the subtasks may be executed to achieve the task.

Hard and soft interactions between tasks, called *NLEs* (non-local effects), are also represented in TÆMS and reasoned about during scheduling and coordination. A complete description of TÆMS is beyond the scope of this paper, however, further background information will be provided where necessary.

A simplified example of a TÆMS task structure for searching the Web for information on *WordPerfect* is shown in Figure 1. The oval nodes are tasks and the square nodes are methods. The top-level task is to `Find-Information-on-WordPerfect` and it has three subtasks: `Query-Infoseek`, `Query-AltaVista`, and `Search-the-Corel-Website` (Corel is the maker of WordPerfect). The top-level task accumulates quality according to the `sum()` qaf so one or more of its subtasks may be performed to satisfy this objective. The `Search-the-Corel-Website` task also has three subtasks: `Find-Corel-URL`, `Best-First-Search-Using-Advanced-Text-Processing`, and `Query-Simple--Corel-Search-Engine`. These tasks are governed by a `max()` qaf thus the quality from any single method determines the quality of the parent task. Note that the expected quality of the `Find-Corel-URL` method is very low relative to the two alternative methods for searching the Corel website; also note the `enables` NLE between the URL finding method and the other methods. These quality attributes and the NLE indicate that finding the URL for Corel is necessary to perform any of the other methods but that it does not contribute much directly to the task of searching the Corel Website, or finding information on WordPerfect, relative to the other two methods. We will return to this example throughout the paper.

[2]The term *task group* is used to denote a set of tasks that are related hierarchy and via non-local effects. Tasks are not joined under a task group if they do not have interactions and are not related hierarchy. A TÆMS model may be comprised of several task groups that are joined under a special *meta* root. In this case the overall objective is to achieve the meta root via the individual task groups, rather than to achieve a single task group via its subtasks.

[3]The number of ways to accumulate quality under the `sum()` qaf is the power-set of its subtasks minus the empty set.

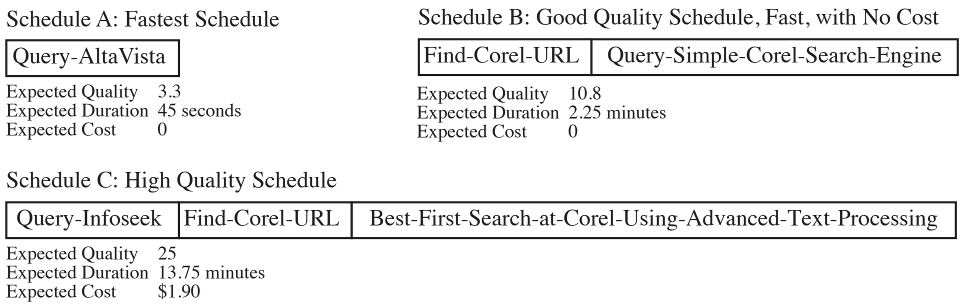


Find-Information-on-WordPerfect
sum()
Query-Infoseek
Quality (30% 0)(70% 10)
Duration (50% 60sec)(25% 180sec) (25% 240sec)
Cost (100% 0)
Query-AltaVista
Quality (40% 0)(50% 5)(10% 8)
Duration (50% 30sec)(50% 60sec)
Cost (100% 0)
Search-the-Corel-Website
max()
Find-Corel-URL
Quality (5% 0)(95% .1)
Duration (50% 30sec)(50% 60sec)
Cost (100% 0)
enables
Query-Simple-Corel-Search-Engine
Quality (10% 0)(90% 12)
Duration (50% 1min)(50% 2min)
Cost (100% 0)
Best-First-Search-at-Corel-Using Advanced-Text-Processing
Quality (10% 0)(90% 20)
Duration (50% 8min)(50% 14min)
Cost (100% $2)
Subtask Relation
Enables NLE
Task
Method

Figure 1: TÆMS Task Structure for Searching for Information on WordPerfect

3 Design-to-Criteria Scheduling

Design-to-Criteria scheduling is the process of finding a satisficing course of action for a complex problem solving activity represented as a TÆMS task structure. The main tenet is to cope with the combinatorial explosion of possibilities while reasoning about quality, cost, and duration criteria such as hard or soft limit/threshold requirements for each dimension and factors describing the relative importance of each dimension. Scheduler client applications or users specify the design criteria and the scheduler *designs* a schedule to best meet the criteria, if possible given the task model. To illustrate this concept, three possible simplified schedules for the WordPerfect search activity are shown in Figure 2. Depending on the design criteria, any one of these schedules could be generated and returned to the client. For instance, if the primary issues are speed and no cost, then the Schedule A is constructed. If time and cost are not at issue then Schedule C is created. If time and cost are important but the client specifies certain quality requirements, then Schedule B is constructed.



Schedule A: Fastest Schedule
Query-AltaVista
Expected Quality 3.3
Expected Duration 45 seconds
Expected Cost 0
Schedule B: Good Quality Schedule, Fast, with No Cost
Find-Corel-URL
Query-Simple-Corel-Search-Engine
Expected Quality 10.8
Expected Duration 2.25 minutes
Expected Cost 0
Schedule C: High Quality Schedule
Query-Infoseek
Find-Corel-URL
Best-First-Search-at-Corel-Using-Advanced-Text-Processing
Expected Quality 25
Expected Duration 13.75 minutes
Expected Cost $1.90

Figure 2: Three Satisficing Schedules

Design-to-Criteria scheduling requires a sophisticated heuristic approach because of the scheduling task's inherent computational complexity. To understand the complexity and get a feel for the scheduling process, consider a task structure only a single level deep, where a single task has m children that are methods and it accumulates quality according to the `sum()` qaf. In this case, there are $2^m - 1$ unordered sets of methods that can be used to achieve the parent task, and within each set of n methods, $n!$ possible orderings of methods in the schedule. Thus, complexity in the scheduling task comes from two sources: the number of unordered method sets that can achieve the high-level objective, $O(2^m)$, and the possible orderings of each method within the schedule, $O(n!)$. This differs from other scheduling tasks where ordering is the primary issue. Clearly, for any significant task structure the brute-strength approach of generating all possible schedules is infeasible.

We partly cope with the computational complexity by using a schedule abstraction called an *alternative*, shown in Figure 3. Alternatives contain sets of unordered methods that can be ordered to form a schedule and an estimate of the quality, cost, and duration distributions that may result from building the schedule. Alternatives are associated with all task nodes in the TÆMS task structure. Alternatives for tasks closer to the root are combinations of the alternatives associated with the subtasks; the subs' alternatives are combined according to the qaf. Thus alternatives are built bottom-up from the leaves to the root. A subset of the alternatives for the root of the task structure are turned into schedules to perform the high level task by achieving the lower level tasks. In a sense, alternatives of the interior nodes represent partial unordered schedules – they describe the actions that can be performed, i.e., methods, to obtain quality for the task node with which they are associated.

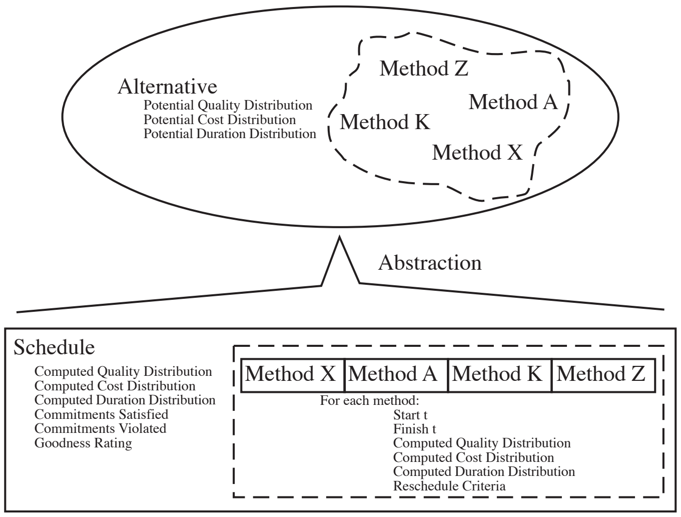

Alternative
Potential Quality Distribution
Potential Cost Distribution
Potential Duration Distribution
Method Z
Method A
Method K
Method X
Abstraction
Schedule
Computed Quality Distribution
Computed Cost Distribution
Computed Duration Distribution
Commitments Satisfied
Commitments Violated
Goodness Rating
Method X
Method A
Method K
Method Z
For each method:
Start t
Finish t
Computed Quality Distribution
Computed Cost Distribution
Computed Duration Distribution
Reschedule Criteria

Figure 3: The Alternative Abstraction

The quality, cost, and duration estimates contained in alternatives describe the potential characteristics of a schedule created from the alternative. The potentials are calculated by ignoring ordering information and applying all NLEs. The alternative abstraction defers the $O(n!)$ ordering component of the complexity problem until schedule time, when orderings are imposed and another technique is used to reduce the $O(n!)$ complexity. However, the alternative abstraction does not address the $O(2^m)$ complexity factor which is driven by the number of possible method sets that can achieve the high-level objective. This complexity is handled dynamically during the alternative generation process. There are exactly three situations that lead to combinatorial explosions of this type:

Build-Up Gradual complexity "build-up" occurs when a task node has a typical number of child nodes, each of which has many alternatives, and the children are joined under the `sum()` qaf.

Complexity of this type is controlled by pruning each alternative set post-hoc. In other words, the gradual build-up is eliminated by pruning the alternative sets of the child nodes after they are generated to keep the number of alternatives, at each step, within a reasonable bounds. The alternatives are pruned according to the quality, cost, and duration criteria specified by the scheduler client and the potentials associated with the alternative. The caveat is that the potential quality, cost, and duration distributions may be overly optimistic depending on the interactions between tasks. This is partly addressed by *improvement* heuristics touched-on later.

Instant Instant complexity problems occur when task node has a large number of child nodes, each with few alternatives, and the children are joined under the `sum()` qaf.

Complexity of this type is predictable by a simple look-a-head operation. In this case, we cannot first generate the alternative set and then prune because the actual number of alternatives that would be generated is too large. Instead, we must heuristically generate a set of alternatives that characterizes the set of possible alternatives with an eye towards the client's quality, cost, and duration criteria. The details of this operation are current research. Note that if the number of child nodes is moderate, we can just generate the alternative set and prune if necessary, as described above, because the exponential factor $O(2^m)$ still translates into a manageable number of alternatives.

Combination Combinations of instant and build-up as described above. This occurs when a task node has a large number of child nodes, each with many alternatives, and the children are joined under the `sum()` qaf. In this case, the solution for the build-up problem above will control the number of alternatives that reside a the child nodes before alternatives are created for the parent node, and the instant solution will control the number of alternatives generated at the parent node.

Thus the $O(2^m)$ type of complexity is controlled by pruning large alternatives sets and heuristically generating partial alternative sets that cannot be exhaustively generated, and subsequently pruned, because of their size. As mentioned previously, not all alternatives associated with the root task node are turned into schedules. Alternatives are selected for scheduling the same way that they are "kept" during the pruning process – by rating alternatives using their potential quality, cost, and duration characteristics according to how well they meet the client's specified criteria.

Once an alternative is selected to be scheduled, a heuristic method rating process is used to determine the proper ordering of alternative methods to create a schedule. The method rating approach controls the $O(n!)$ complexity by not generating all possible orderings of the methods. Methods are rated using the types of heuristics described below. While the complexity of some of the method rating heuristics is polynomial in the number of task structure nodes, overall the savings of this approach versus the $O(n!)$, $\omega(2^n)$, possible orderings is pronounced and makes the problem tractable.

- Enforce hard NLEs, i.e., enforce precedence constraints.
- Enforce earliest start times and deadlines.
- Try to take advantage of positive soft NLEs, where doing one activity before another improves overall utility.
- Try to avoid negative soft NLEs, where doing one activity before another degrades overall utility.
- Try to satisfy external commitments and avoid violating them.
- Try to improve overall schedule quality quickly - a greedy heuristic.

While the method rating heuristics in place today used fixed trade-off information, such as deadlines being more important than commitment satisfaction, future method rating heuristics will utilize client specified criteria where appropriate. This will enable clients to specify things like "meeting commitment C is more important that anything else, including quality, cost, and duration." The satisficing approach will be similar to that used to handle satisficing at the alternative and schedule level with respect to quality, cost, duration, thresholds and uncertainty [20].

After each schedule is generated (all methods are added or discarded due to constraints) it is critiqued by a set of improvement heuristics that ascertain if adding other methods or alternatives to the schedule will improve its overall quality, cost, and duration characteristics. Typically, the critics look for methods that are positively affected by hard or soft interactions that are missing from the current schedule. For example, if performing method `A` may improve the quality and reduce the cost of method `B`, but method `A` is omitted from a schedule that includes method `B`, it may be worthwhile to add `A`. Improvements are suggested to the system not by tweaking the schedule at hand, but by suggesting an alternative that includes the items lacking in the current schedule. The new alternative is added to the set of eligible alternatives and selected, or not, according to how well its potentials meet the client's criteria relative to the other alternatives.

The process of selecting alternatives and building schedules iterates until the number of schedules crosses a threshold, all the alternatives are scheduled, or the remaining alternatives cannot lead to better schedules (determined by the alternatives' potentials). Schedules are then rated against the client's quality, cost, and duration criteria and the best one is returned for execution. This iterative process has an anytime [22, 21] flavor since generating the set of alternatives for the root task is relatively cheap complexity-wise and fast real-timewise, due to complexity control, compared to the process of building schedules. The scheduler can generate a small set of schedules quickly, but given more time, it can explore more schedules and increase the probability that a "better" schedule is created.[4]

It is important to recognize that the scheduling process is typically not a one-shot activity. Because of the uncertainty involved, it is entirely possible for a method execution to return results outside of the bounds of expectations thus requiring rescheduling. However, in situations where rescheduling is not appealing, the scheduler can create less efficient schedules that are more fault tolerant by making more conservative probabilistic assumptions about quality, cost, duration and uncertainty. This relates somewhat to previous work done by Durfee and Lesser [11] in which schedules are made "loose" by increasing duration expectations when building schedules, effectively creating a slack-time buffer between each action. Our model is much stronger in that we change expectations based on probabilities rather than using magic numbers, and we do so in all dimensions, quality, cost, and duration.

4 Intermittent Processing

Formerly, TÆMS methods have been viewed as actions that require all the computational resources of the agent for which they are scheduled. We have extended the method model to describe actions that do not require 100% of the agents' resources during their execution intervals. This allows us to accurately model activities that contain interleaved I/O, like interacting with an external sensor or interleaving `http get` requests and local processing as shown in Figure 4. Note that we do not model *when* the delays occur during execution because TÆMS makes no assumptions about

[4]If the potential distributions for quality, cost, and duration that are contained in the alternatives are good indicators of the schedule quality, then the algorithm will produce "good" schedules from the start and adding time only increases the certainty that "better" schedules will not be generated.

the execution characteristics of methods during their time intervals (methods are black boxes)[5]. Another reason not to model when the delays occur is simply that in some cases it is difficult to predict when a particular action will engage in an activity that causes delays, i.e., it might be dependent on the inputs. We also do not assume or require that the delays actually take place each time the action is performed[6]. However, if the action is likely to have internal delays, and this can be described statistically, then the embedded delay model is appropriate.

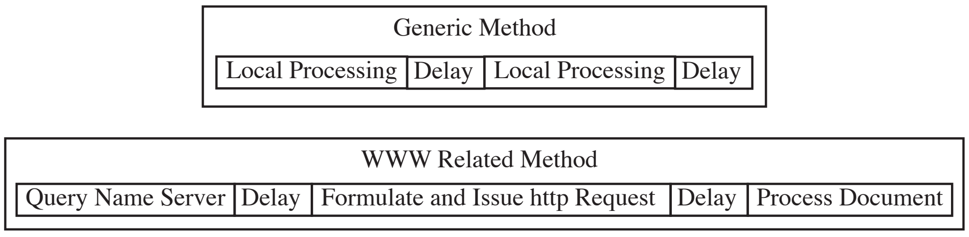


Figure 4: Embedded Delays

In addition to embedded delays, the enhanced TÆMS model also supports external delays that arise when the results from one processor must be communicated to another spatially distributed processor. Note that the processors may be agents or processors that are working on a common task or external resources that exhibit the same delayed propagation of results. Figure 5 illustrates this concept.

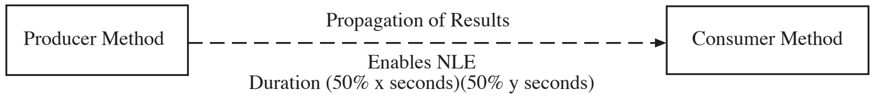


Figure 5: External Delays

4.1 Embedded Delays

Actions with embedded delays utilize only a portion of the processor during their execution interval. Embedded delays are expressed in TÆMS by noting that a particular method uses only a certain percentage of the processor during its execution. Literally, a simple `utilization factor` is associated with each method.

Since the processor's capacity is under utilized during the execution interval of methods that contain embedded delays, and one of the primary objectives in Design-to-Criteria scheduling is to meet soft real-time deadlines, it is advantageous to attempt to use the unused processing resource by performing other actions during the interval containing the delay. This "doubling up" of methods is different than the previous concurrent TÆMS related scheduling work [17, 12] in which schedules are constructed for multiple processors or multiple lines of control. The most important difference between the models is not the issues they must address, but rather their applicability

[5]For clarity, consider the issue of when methods actually accumulate quality. The black box assumption means that even though methods may accumulate quality linearly during execution, or during the last moment of method performance, the model regards methods as producing quality only when they are completed. This is the most general assumption possible with respect to methods and their execution behaviors – it enables us to model a wide range of activities using the same modeling construct. However, the assumption sacrifices some degree of modeling precision to obtain this generality.

[6]If the delays do not occur, however, the execution performance of the schedule will not adhere to the predictions made by the scheduler, i.e., it must assume that the delays will occur and act accordingly.

to certain domains or certain classes of task structures. To effectively utilize multiple processors, task interactions must fall in such a way that processors can work concurrently in an independent fashion most of the time. In other words, the interactions between tasks cannot lead to a serialization, or partial serialization, of tasks or it is difficult to use multiple processors effectively. On the other hand, the single processor "doubling up" approach is useful for task structures that contain task interactions because it is easier to keep the single processor busy doing useful work. To examine the difference from an analytical perspective: creating n efficient parallel schedules requires n asynchronous lines of control, creating 1 parallel schedule requires only 1 line of control which can be mostly synchronous if embedded delays are to be utilized, or completely synchronous if not. Thus, modeling embedded delays and scheduling for them promotes a good balance between resource efficiency and performance when task structures are moderately complex.

In terms of the alternative generation process, the precursor to scheduling, embedded delays are handled automatically by the alternative abstraction, which is to say, they are ignored. This is deliberate and in keeping with the objective of alternatives as cheap to compute schedule abstractions. It is necessary to ignore embedded delays because it cannot be determined prior to building the schedule whether or not the unused processing power can be utilized by other methods. To the extent that the unused processing power can be utilized by the scheduling algorithm, the estimated duration associated with alternatives that contain methods with embedded delays will be an overstatement. Overestimation occurs because durations of methods with delays, by necessity as stated, are treated at face value.

To utilize the unused processing power generated by embedded delays, our scheduling solution relies on one assumption, actions are interruptible and there is a mechanism for doing so. Recall that the model does not express when the delays occur. Consequently, the "interruptibility" assumption is necessary and sufficient to take advantage of the unused processing resources[7] A sketch of the proof follows.

The invariant is processor load, i.e., at no time can the processor actually execute more than one method. Let $Requires(x)$, x is a method, denote the amount of time required to complete method x including the time used by any embedded delays. Let $Unused(x)$, x is a method, denote the amount of execution time of method x during which the processor is unused. For methods that do not contain embedded delays, $Unused(x) = 0$. Let M_d denote a method containing delays. Let M_{nd} denote a method that contains no delays.

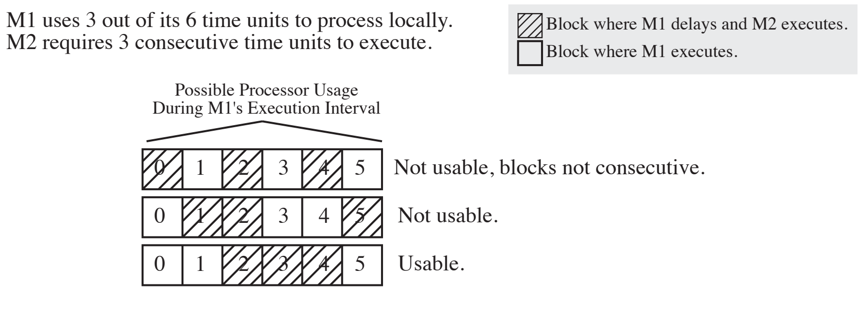

M1 uses 3 out of its 6 time units to process locally.
M2 requires 3 consecutive time units to execute.
▨ Block where M1 delays and M2 executes.
☐ Block where M1 executes.
Possible Processor Usage During M1's Execution Interval
0 1 2 3 4 5
Not usable, blocks not consecutive.
Not usable.
Usable.

Figure 6: Overlap Difficult to Utilize without Interruptible Methods

Assume that methods are not interruptible. Consider a simple case where $Unused(M_d) = Requires(M_{nd})$, that is the processing resources unused during the execution of M_d are exactly

[7]This interruptibility assumption is in keeping with the general black box view of methods in TÆMS. We do not assume, for example, that methods are now anytime and may be interrupted and still produce a result. A method that is interrupted is simply stopped in time – if it does not resume and complete execution, no results will be produced.

sufficient to cover the resources required by M_{nd}. To take advantage of the unused processing resources, we must overlap M_{nd} and M_d. However, since methods are not interruptible and we have no guarantees about when M_d's delays occur, there are many possible execution traces in which M_{nd} does not get enough consecutive time slots to execute, illustrated in Figure 6. To quantify the problem for this simple case, assuming time is an integer, the number of ways that the delays can fall to yield a consecutive block large enough to execute M_{nd} without interrupting either method is $Requires(M_d) - (Unused(M_d) - 1)$. The number of possible combinations for the delays if they fall randomly is $\begin{pmatrix} Requires(M_d) \\ Unused(M_d) \end{pmatrix}$, thus the probability of randomly getting a sufficient consecutive block, Equation 1, can be very small indeed.

$$\frac{Requires(M_d) - (Unused(M_d) - 1)}{\begin{pmatrix} Requires(M_d) \\ Unused(M_d) \end{pmatrix}} = \frac{Requires(M_d) - (Unused(M_d) - 1)}{\frac{Requires(M_d)!}{Unused(M_d)!(Requires(M_d) - Unused(M_d))!}} \tag{1}$$

Given the assumption that methods are interruptible, we overlap methods during scheduling following a straightforward algorithm. When scheduling a method, we determine the amount of unused processor time in the last method that was added to the schedule. If there is unused processor time, there are two basic cases for overlapping.

1. The method being scheduled can be performed entirely within the time boundaries defined by the previous method(s). Recall that we do not assume that delays are uniformly distributed during execution time nor do we make any assumptions about where the delays may fall. Thus, if the method currently being considered "fits" within the previous interval, we insert it there and adjust the remaining slack time for that interval accordingly.

   To determine if a method will fit within the previous interval, we compute the method's finish time without utilizing the slack time in the previous interval, then subtract the slack available in the previous interval. If this brings the method's finish time into the previous interval, we know that $Requires(M_{new})$ is less than or equal to the slack in the previous interval. Note, the computation is more straightforward in the form of $if\ Requires(M_{new}) <= Unused(M_{prev})$. The longer form is useful as it also determines the finish time of M_{new} if it must go beyond the first interval (the following case).

2. The other case is that the finish time of M_{new} must extend beyond M_{prev}. This is because either there is insufficient slack left during M_{prev} or simply because $Requires(M_{new}) > Requres(M_{prev})$. In either case, because M_{new} may also contain embedded delays, and we cannot make assumptions about these, just as we cannot make assumptions about M_{prev}, we must overlap the portion of M_{new} into the previous interval, and then extend beyond the previous interval by the quantity of $Requires(M_{new})$ that will not fit into the previous interval. This introduces an apparent inefficiency into the overlapped methods, however, to compensate, if there is slack time left, though we cannot reason about where it falls, we know it falls during the interval between M_{new}'s starting time and M_{new}'s adjusted finish time.

Note, this discussion is cast in terms of expected values. The probabilistic case is more complicated. The cases above are pictured algorithmically in Figure 7. To illustrate case two, consider a

```
// Let Mn be the new method being added to the schedule.
// Let Mp be the previous method on the schedule.
//

if ( this_is_the_first_element == true || slack_time_in_prev == 0 )

    append Mn to the schedule
    slack_time_in_next_interval = Unused( Mn )

else

    Mn_candidate_finish_time = Mp_finish_time + Requires( Mn ) - slack_time_in_previous

    if ( Mn_candidate_finish_time <= Mp_finish_time )

        // it fits, but we cannot make assumptions about where delays fall.  endtimes are the same.
        //
        Mn_start_time = Mp_start_time
        Mn_finish_time = Mp_finish_time
        slack_time_in_next_interval = slack_time_in_prev - Requires(Mn) + Unused(Mn)

    else
        // it doesn't fit, we must extend beyond region of Mp.
        //
        Mn_start_time = Mp_start_time
        Mn_finish_time = Mn_candidate_finish_time
        length_of_extension = Mn_finish_time - Mp_finish_time

        // we don't know when the delays fall, but, we know how much is being used over the whole interval.
        //
        slack_time_in_next_interval = slack_time_in_prev + length_of_extension -
                    ( Requires(Mn) - Unused( Mn ))

    endif

endif
```

Figure 7: Algorithm for Handling Embedded Delays when Scheduling a Method (Expected-Value Representation)

situation in which there are three methods being scheduled, M1, M2, and M3. Say each method requires 5 time units to execute and each method uses only 10% of the processor during its execution period.

- Method M1 will be scheduled from 0 to 5, and there will be 4.5 units of unused processor time during that interval.
- Method M2 will be scheduled from 0 to 5.5 rather than 0 to 5. Even though M2 only needs .5 units of processor time, we cannot be certain that M2's processor needs will fall at a different time than M1's, thus we must extend M2's finish time beyond the end of the interval by that amount. However, we record the fact that during the 0 to 5.5 interval, there are 4.5 units of unused processor time.
- Method M3 will be scheduled from 0 to 6 for the same reasons that M2 was extended. During this interval, there are still 4.5 units of unused processing time.
- If a fourth method, M4, arrives that requires 5 units of processing time and uses 100% of the processor during this time, M4 will also start at time 0 and will end at 6.5, and there will be no slack remaining during the 0 to 6.5 interval.

Observant readers will notice that in the first case, where a method's duration requirements completely fit within the previous method's slack time, any slack time of the method being scheduled is added to the pool of slack time for that interval. The reason for using requirements, rather than $Requires(m) - Unused(m)$ to determine if a method will fit into existing slack time is that a method's duration requirements are considered hard specifications, i.e., under no circumstances can methods be put into intervals smaller than their $Requires(m)$ distribution. From an analytical perspective, if we remove that constraint and overlap using the $Requires(m) - Unused(m)$ factor, it is possible for the delays of the methods sharing an interval to overlap resulting in no processing being done on any of the methods and correspondingly execution times violating expectations.

In addition to affecting the manner in which methods get added to the schedule, we must also consider embedded delays whenever we ask "what-if" type questions about when a method is expected to start or finish. The issue of embedded delays is ubiquitous in the method rating functions described in Section 3. Design and implementation wise, the change is less pronounced than it sounds. The class responsible for maintaining and reasoning about schedules considers embedded delays when answering questions posed by the method rating functions. Supporting embedded delays also does not increase the complexity of any of the method rating functions as the expense is dominated by other factors.

4.2 External Delays

Unlike embedded delays that are associated with method internals, external delays are associated with NLEs between tasks or methods. In other words, when one action is dependent on the results of some other action or task, it is represented explicitly in TÆMS using either hard or soft NLEs. If appreciable communication time is required to send the results it is represented as a propagation delay, expressed statistically as a discrete probably distribution, associated with the NLE arc.

As with embedded delays, external delays are handled automatically with respect to alternative generation and estimates – the potentials or estimates for quality, cost, and duration, are computed without regard for propagation delays. External delays, like other time related constraints, e.g., earliest allowable start times, can affect the estimates associated with alternatives by causing their

```
Determine-Earliest-Time-Method-May-Be-Scheduled
    gather-hard-nles-with-delays and store in hard-delay-set
    //
    // We must wait until all results associated with hard NLEs have time
    // to reach this method.
    //
    latest-hard-delay = find-latest-hard-delay( hard-delay-set )
    if ( latest-hard-delay > earliest-start-time )
        earliest-method-may-be-scheduled = latest-hard-delay
    else
        earliest-method-may-be-scheduled = earliest-start-time
```

Figure 8: Determining Earliest Time a Method may be Considered

durations to be understated. Underestimates in duration occur with this class of constraints because idle time may have to be inserted into the schedule to enforce the time constraints, i.e., an agent may be forced to go idle while waiting for results from elsewhere. However, the underestimation of duration will decrease to the extent that the time spent waiting for results can be used by other processing activities.

To schedule for models that are affected by external delays, the scheduling algorithm must defer the execution of methods that are on the receiving end of the delay at least until the expected results have time to propagate. This is enforced by the method rating heuristics. Methods that are on the receiving end of hard NLEs with delays cannot be scheduled until the results have time to propagate, that is they are rated in such a way to remove them from the candidate set of methods until the delay time has passed. In keeping with the overall collage' of rating heuristics, methods that benefit from delayed soft NLEs are not deferred solely to take advantage of the NLEs. When a method is affected by an external delay, the earliest "time" at which the method will again be considered for scheduling is either the greatest of its hard delays or its earliest start time, whichever is greater, described algorithmically in Figure 8. As methods that are on the receiving end of hard external delays are deferred, methods that are on the sending end of the data are given preferential treatment, i.e., they are scheduled as early as possible given all other scheduling constraints and considerations. This too is handled by method rating heuristics.

The added advantage of handling external delays via the method rating heuristics is that the gap required between a consumer method and its producer method is automatically filled with the best available method or methods. In the event that no methods are available, due to various other constraints on the remainder of the method set, slack time is inserted. As with other time related constraints, e.g., earliest start times, it is possible to schedule combinations of methods and slack time elements during an external delay interval depending on other scheduling constraints. It is also likely for the items "plugged in" to the delay interval to take more time than strictly required by the interval – the automatic stretching of the interval is another feature of integrating external delays with the other method rating mechanisms. Figure 9 illustrates this feature. M_P is a producer method, M_C is a consumer method, and they are joined by a hard NLE that includes a delay distribution, i.e., it takes time for the results to propagate. The figure shows a conceptual schedule and various possible uses of the local processing capability during the propagation delay.

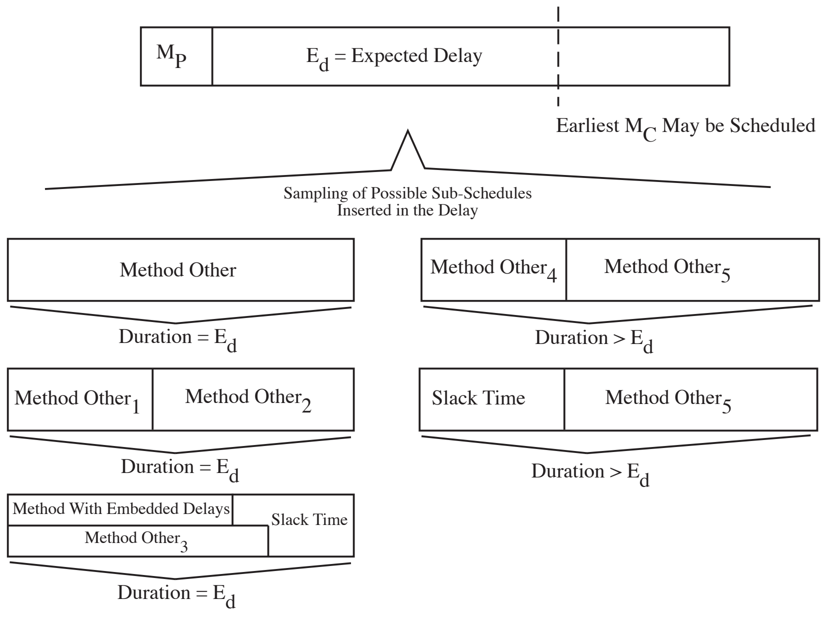


Figure 9: Various Approaches in which External Delays are Utilized

5 An Example

In this section we revisit the TÆMS task structure from Section 2. Appropriate embedded delays have been associated with all methods, Figure 10, as they all involve embedded I/O. In the case of the `Query-Infoseek` and `Query-AltaVista` methods, the embedded delays represent the expected amount of time spent waiting during the interval from when the internal `http` requests are issued and answers are received. The longer duration and the lower local CPU utilization factor of `Query-Infoseek` models the relatively slow response from the Infoseek search engine. The AltaVista site, on the other hand, tends to respond in a more timely fashion thus the overall duration of `Query-AltaVista` is less and less time is expected to be spent waiting for a response. However, quality is independent from duration – the higher expected quality for the Infoseek related method models the expectation that the Infoseek information retrieval engine will do a better job of finding relevant URLs than the AltaVista engine.

The methods related with the `Search-the-Corel-Website` task likewise have embedded delays. The `Find-Corel-URL` method's embedded delays reflect the time spent waiting for a response from a remote name server or remote URL knowledge base. The delays associated with `Query-Simple-Corel-Search-Engine` are related to expected delays in getting information from the Corel site. Similarly with the embedded delays of the `Best-First-Search-Using-Advanced--Text-Processing` method, however, as this method requires the services of another location's text processing system, and it performs many separate `http get` requests from the Corel site, this method's CPU utilization factor is much lower. In other words, because of all the network related activity and remote processing this method will actually use very little of the local CPU. Note that the quality expectations for the two search methods also vary. The method that uses the advanced text processing techniques is expected to return much better results, but also to take much longer and to incur a service fee from the technology source.

This example also contains external delays. The methods that rely on the Corel URL must wait a small amount of time for the results to propagate from the method that obtains the URL.

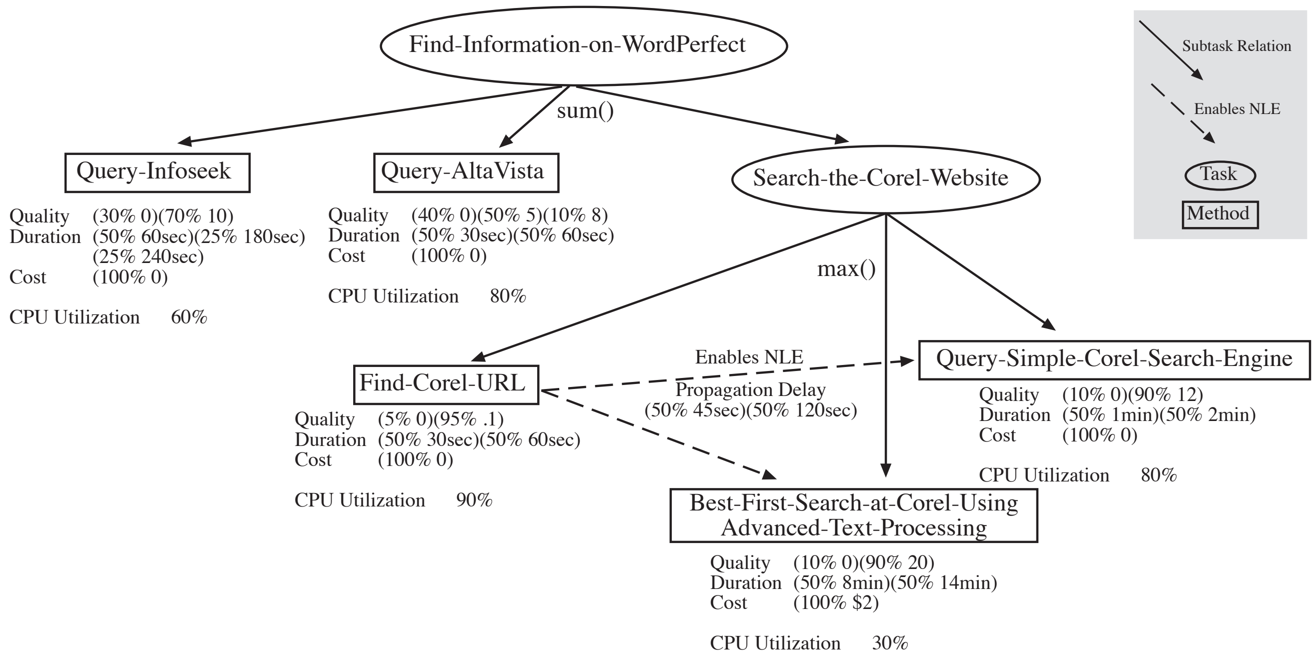


Figure 10: Find Information Task Structure with Delays

This relationship is expressed as an *enables* NLE from the producer method to the consumers.

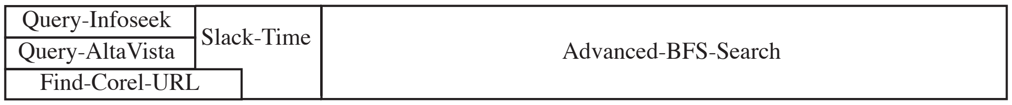
A simplified Design-to-Criteria generated schedule for this task structure, given criteria that specifies quality is the paramount objective, is shown in Figure 11. The embedded delays in `Query-Infoseek` and `Query-AltaVista` allow them to be overlapped entirely. As the unused processing time during the interval in which these methods execute is insufficient to execute `Find-Corel-URL`, its finish time is extended beyond the end of the interval. Since the URL found by `Find-Corel-URL` requires time to propagate, and there are no other methods to be performed, slack time is inserted before `Best-First-Search-Using-Advanced-Text-Processing`. It is interesting to note that the scheduler did not choose to execute the `Find-Corel-URL` method first, and then overlap the other two search engine methods during the URL finding interval and the subsequent delay interval. This is because one of the method rating heuristics is greedy – it attempts to get some quality to the root task as soon as possible.[8] In this particular case, the various method constraints fell in such a way that the greedy heuristic dominated the heuristics that prefer to schedule producer/enabling methods earlier in the schedule.

Another schedule for a different set of criteria is shown in Figure 12. This schedule is the best given the objective of keeping down the cost while obtaining good quality and not being too concerned about duration. Since the advanced text processing method incurs a charge, and takes a very long time, the best schedule does not use that method but instead uses the free `Query-Simple-Corel-Search-Engine`. The initial portion of the schedule is unchanged as the constraints governing method placement are also unchanged.

Given the objective of finding information very quickly without spending any money, the scheduler will produce the schedule found in Figure 13. The expected quality of this schedule is less than half of the expected quality for the schedule in Figure 12, however, the duration is greatly reduced. The fact that two methods are scheduled, rather than just one of `Query-Infoseek` or

[8]In uncertain environments, obtaining some quality or some solution as quickly as possible is not an unreasonable approach.

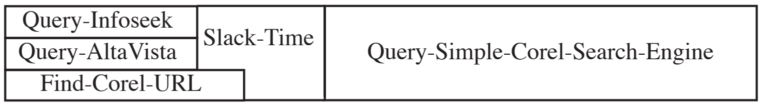
Conceptual Criteria: Quality Importance High, Cost Importance NIL, Duration Importance NIL



Schedule Quality: (15% 0)(50% 30)(30% 35)(6% 38), Expected Value = 27
Schedule Cost: (5% 0)(95% 2.0), Expected Value = $1.90
Schedule Duration: (5% 154sec)(47% 844sec)(24% 994sec)(24% 1204sec), Expected Value = 871 seconds

Figure 11: Best Schedule where Quality is the Emphasis

Conceptual Criteria: Quality Importance High, Cost Importance High, Duration Importance Low

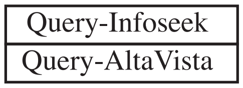


Schedule Quality: (13% 0)(51% 22)(30% 27)(6% 30), Expected Value = 21
Schedule Cost: (100% 0), Expected Value = $0.00
Schedule Duration: (2% 154sec)(18% 214sec)(44% 274sec)(36% 484sec), Expected Value = 329 seconds

Figure 12: Best Schedule for High Quality, Zero Cost, and Some Weight on Duration

`Query-AltaVista`, may seem troubling. However, this is not an error. Because both methods contain embedded delays they can be executed during the same interval with no addition to expected duration, modeling the embedded delays obtains two method executions for the duration of one and increases quality to boot. A lower ranked candidate schedule produced with this criteria, that contains just the execution of `Query-Infoseek`, returns an expected quality value of seven in contrast to this schedule's expected value of ten.

Conceptual Criteria: Quality Importance Low, Cost Importance High, Duration Importance High



Schedule Quality: (20% 0)(39% 10)(35% 15)(7% 18), Expected Value = 10
Schedule Cost: (100% 0), Expected Value = $0.00
Schedule Duration: (50% 60sec)(25% 180sec)(25% 240sec), Expected Value = 135 seconds

Figure 13: Best Schedule Given "Cheap and Fast" Criteria

6 Conclusion

Enhancing the TÆMS task model to represent embedded and external delays improves its ability to represent particular classes, e.g., network or sensor related, of problem solving activities. The CPU utilization factor used to denote the occurrence of embedded delays supports better processor utilization by the Design-to-Criteria scheduler, which can then overlap or multiplex multiple methods during a given execution interval. When overlapping methods, particular care must be paid to start and finish times, and any remaining unused processor resource, to guarantee that load does exceed 100% at any given time. As discussed, violating the 100% invariant during scheduling translates into method overlap failure at execution time, resulting in the multiplexed methods taking more time to execute. In contrast to previous concurrent TÆMS scheduling approaches, the small concurrency provided by doubling up methods appears to be the right grain-size for the types of problems (having interactions) modeled with TÆMS.

The external delays enhancement allows the scheduler to explicitly reason about the amount of time required for results or messages to propagate from one task to another. Without modeling this class of delays, the scheduler would have no choice but to create schedules destined for duration overrun as the propagation delay component would be "hidden" from the scheduler's view. However, the model enhancement does more than lead to better duration expectations. Because it describes formerly hidden durations that do not use the local processing resource, other methods may be executed during this interval. The process of selecting which methods to execute during an external delay interval is handled automatically integrating the external delay support into the existing Design-to-Criteria method rating heuristics.

Future work on Design-to-Criteria scheduling and the TÆMS task model lies in the areas of satisficing method selection and runtime negotiation between the scheduler and its client. A satisficing approach to method selection, discussed in Section 3, would allow clients to express goals like "for this task, satisfying commitment C is twice as important as keeping down the cost." Currently, the relative importance of the different method rating heuristics is built-in to the scheduler and not specified by the client.

Due to the computational complexity of determining a course of action for a given TÆMS task structure, client's often do not know a priori what classes of solutions are possible. Runtime negotiation between the scheduler and the client would help refine the client's goal criteria based on what is actually possible given the task structure at hand. In a situation where the even a reasonable approximation of the client's goal criteria cannot be found (though the scheduler will return the best possible satisficing schedule), the client may decide to change the goal criteria and try again. Interleaving scheduling and criteria refinement would make this entire process more more efficient as the changing criteria would dynamically target the scheduler at the client's evolving satisficing goal.

References

[1] N. Carver and V. Lesser. The dresun testbed for research in fa/c distributed situation assessment: Extensions to the model of external evidence. In *Proceedings of the International Conference on Multiagent Systems*, June, 1995.

[2] Keith S. Decker. *Environment Centered Analysis and Design of Coordination Mechanisms.* PhD thesis, University of Massachusetts, 1995.

[3] Keith S. Decker. TÆMS: A framework for analysis and design of coordination mechanisms. In G. O'Hare and N. Jennings, editors, *Foundations of Distributed Artificial Intelligence*, chapter 16. Wiley Inter-Science, 1995. Forthcoming.

[4] Keith S. Decker and Victor R. Lesser. Quantitative modeling of complex computational task environments. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 217–224, Washington, July 1993.

[5] Keith S. Decker and Victor R. Lesser. Quantitative modeling of complex environments. *International Journal of Intelligent Systems in Accounting, Finance, and Management*, 2(4):215–234, December 1993. Special issue on "Mathematical and Computational Models of Organizations: Models and Characteristics of Agent Behavior".

[6] Keith S. Decker and Victor R. Lesser. Designing a family of coordination algorithms. In *Proceedings of the Thirteenth International Workshop on Distributed AI*, pages 65–84, Seattle, WA, July 1994. AAAI Press Technical Report WS-94-02. Also UMass CS-TR-94-14. To appear, Proceedings of the First International Conference on Multi-Agent Systems, San Francisco, AAAI Press, 1995.

[7] Keith S. Decker and Victor R. Lesser. Experimenting with a family of coordination algorithms. Submitted to IJCAI-95, 1994.

[8] Keith S. Decker and Victor R. Lesser. Task environment centered design of organizations. In Ingemar Hulthage, editor, *Computational Organization Design*. AAAI Spring Symposium, 1994. Working Notes.

[9] Keith S. Decker and Victor R. Lesser. Coordination assistance for mixed human and computational agent systems. In *Proceedings of Concurrent Engineering 95*, pages 337–348, McLean, VA, 1995. Concurrent Technologies Corp. Also available as UMASS CS TR-95-31.

[10] K.S. Decker, V.R. Lesser, M.V. Nagendra Prasad, and T. Wagner. MACRON: an architecture for multi-agent cooperative information gathering. In *Proccedings of the CIKM-95 Workshop on Intelligent Information Agents*, Baltimore, MD, 1995.

[11] E. Durfee and V. Lesser. Predictability vs. responsiveness: Coordinating problem solvers in dynamic domains. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 66–71, St. Paul, Minnesota, August 1988.

[12] S. Fujita and V. Lesser. Cooperative tasks in coarse grain search problems. CS Technical Report 96–28, Univ. of Massachusetts, 1996.

[13] Alan Garvey and Victor Lesser. Design-to-time real-time scheduling. *IEEE Transactions on Systems, Man and Cybernetics*, 23(6):1491–1502, 1993.

[14] Alan Garvey and Victor Lesser. Design-to-time scheduling with uncertainty. CS Technical Report 95–03, University of Massachusetts, 1995.

[15] Alan Garvey and Victor Lesser. Representing and scheduling satisficing tasks. In Swaminathan Natarajan, editor, *Imprecise and Approximate Computation*, pages 23–34. Kluwer Academic Publishers, Norwell, MA, 1995.

[16] Alan Garvey and Victor Lesser. Issues in design-to-time real-time scheduling. In *AAAI Fall 1996 Symposium on Flexible Computation*, November 1996.

[17] Qiegang Long and Victor Lesser. A heuristic real-time parallel scheduler based on task structures. CS technical report, University of Massachusetts, 1995. In preparation.

[18] Tim Oates, M. V. Nagendra Prasad, and Victor R. Lesser. Cooperative information gathering: A distributed problem solving approach. Technical Report 94-66, Department of Computer Science, University of Massachusetts, September 1994.

[19] Tim Oates, M. V. Nagendra Prasad, Victor R. Lesser, and Keith S. Decker. A distributed problem solving approach to cooperative information gathering. In *AAAI Spring Symposium on Information Gathering in Distributed Environments*, Stanford University, March 1995.

[20] Thomas Wagner, Alan Garvey, and Victor Lesser. Satisficing evaluation functions: The heart of the new design-to-criteria paradigm. In *UMASS Department of Computer Science Technical Report TR-1996-82*, November, 1996.

[21] Shlomo Zilberstein. Operational rationality through compilation of anytime algorithms. Ph.D. Dissertation, Department of Computer Science, University of California at Berkeley, Berkeley, CA, 1993.

[22] Shlomo Zilberstein and Stuart J. Russell. Constructing utility-driven real-time systems using anytime algorithms. In *Proceedings of the IEEE Workshop on Imprecise and Approximate Computation*, pages 6–10, Phoenix, AZ, December 1992.