function rather than a fixed output specification[3].

One exception is the Concord system, developed by Lin *et al.* [1987]. Concord is a programming language that supports approximate computations in which the run-time of the subroutine is controlled by the consumer of the results. The main design issues involve the run-time environment structures needed to support flexible procedure calls. Its development was motivated, like our compilation scheme, by the problem of optimizing performance given a limited supply of system resources. For each procedure, a supervisor is used to record values of the approximate results obtained to date, together with a set of error indicators. When a procedure is terminated, its supervisor returns the best result found. Intermediate results are handled by the caller using a mechanism similar to exception handling. The handlers for imprecise results determine whether a result is acceptable or not; this decision is local to the caller, rather than being made in the global utility context that we use. In this sense, Concord actually performs satisficing rather than optimization. Concord has several other disadvantages compared to our approach: it leaves to the programmer the decision of what quality of results is acceptable; it does not mechanize the scheduling process but only provides tools for the programmer to perform this task; and it does not provide for simple cumulative development of more complex anytime systems.

5.2 Further work

There is still much system work to be done in order to implement a complete set of compilation methods as an integral part of a programming language for anytime computation. We also need to understand how best to represent multidimensional, probabilistic and conditional performance profiles, and how to insert monitors to check the partial results obtained and update the PPs accordingly.

We are currently extending the framework to cover the generation and scheduling of *anytime actions* and *observation actions*, both of which are essential for the construction of autonomous agents. Anytime actions are actions whose outcome quality improves gradually over time. For example, moving toward a target in order to get a better view is an interruptible anytime action. Aiming a gun at a target is another example of an interruptible anytime action. In many cases anytime actions can be implemented by interleaving computation and action. Our ultimate goal in this project is to construct a real-time agent that acts by performing anytime actions and makes decisions using anytime computation.

References

[Boddy and Dean, 1989] M. Boddy and T. Dean. Solving time-dependent planning problems. Technical Report CS-89-03, Department of Computer Science, Brown University, Providence, 1989.

[Dean and Boddy, 1988] T. Dean and M. Boddy. An analysis of time-dependent planning. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 49–54, Minneapolis, Minnesota, 1988.

[Elkan, 1990] C. Elkan. Incremental, approximate planning: abductive default reasoning. In *Proceedings of the AAAI Spring Symposium on Planning in Uncertain Environments*, Palo Alto, California, 1990.

[Haussler, 1990] D. Haussler. Probably approximately correct learning. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 1101–1108, Boston, Massachusetts, 1990.

[Horvitz, 1987] E. J. Horvitz. Reasoning about beliefs and actions under computational resource constraints. In *Proceedings of the 1987 Workshop on Uncertainty in Artificial Intelligence*, Seattle, Washington, 1987.

[Korf, 1988] R. Korf. Real-time heuristic search: new results. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 139–144, Minneapolis, Minnesota, 1988.

[Laffey *et al.*, 1988] T. J. Laffey, P. A. Cox, J. L. Schmidt, S. M. Kao and J. Y. Read. Real-time knowledge based systems. *AI Magazine*, 9(1):27-45, Spring 1988.

[Lin *et al.*, 1987] K. J. Lin, S. Natarajan, J. W. S. Liu and T. Krauskopf. Concord: A system of imprecise computations. In *Proceedings of COMPSAC '87*, pages 75-81, Tokyo, Japan, October 1987.

[Michalski and Winston, 1986] R. S. Michalski and P. H. Winston. Variable precision logic. *Artificial Intelligence*, 29(2):121-146, 1986.

[Russell and Wefald, 1989] S. J. Russell and E. H. Wefald. Principles of metareasoning. In R.J. Brachman *et al.*, (Eds), *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, Morgan Kaufmann, San Mateo, California, 1989.

[Vrbsky *et al.*, 1990] S. V. Vrbsky, J. W. S. Liu and K. P. Smith. An object-oriented query processor that returns monotonically improving approximate answers. Technical Report UIUCDCS-R-90-1568, University of Illinois at Urbana-Champaign, 1990.

[Zilberstein, 1990] S. Zilberstein. Compilation of anytime algorithms. Research Proposal, University of California, Berkeley, November 1990.

[3] In fact, the notion of anytime applies much more broadly, for example to contracts among economic agents. A crude version is presented by the model ranges offered by car and computer manufacturers, where different allocations of money will obtain different quality of results. We are beginning to investigate the economics literature to see if similar generalizations have been proposed.

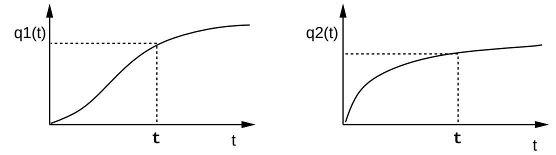

Figure 3: Compilation of a sequence

(since we can use the best solution among the alternatives) and the given amount of time should be allocated to the component that has higher expected quality. This is essentially the case studied by Dean and Boddy [1988]. Rather than using a generalized sequence construct and a particular quality combination function, it might be appropriate to supply a special construct `any` for the case of multiple alternative methods.

4.2.2 Conditional statement

Consider a real-time currency trading program that uses one of two different trading strategies (A and B) depending on whether interest rate will rise (P). We would represent this by the conditional statement:
(`if` $\mathcal{P}$ `then` $\mathcal{A}$ `else` $\mathcal{B}$)
Conditional statements have several variations depending on whether the condition $\mathcal{P}$ is calculated by an anytime algorithm or whether there is a penalty, over and above the cost of time, for executing $\mathcal{A}$ when the condition is false. Here we analyze the compilation for the case in which $\mathcal{P}$ is a fixed-time algorithm that returns (after time $t_{\mathcal{P}}$) the probability (p) that the condition is true. We also assume that the overall quality is the quality of $\mathcal{A}$ when the condition is true and the quality of $\mathcal{B}$ when the condition is false. The optimal time allocation is given by:

$$\max_{0 \le x \le t - t_{\mathcal{P}}} \{p q_{\mathcal{A}}(x) + (1-p) q_{\mathcal{B}}(t - t_{\mathcal{P}} - x)\}$$

A partially evaluated version of this expression is inserted into the 'object code', to be evaluated at run-time when the value of p is known. A PP can be computed at compile time based on the *a priori* value of p.

4.2.3 Loops

Any system that repeatedly performs a complex task can be implemented using a loop through a sequential anytime process. Examples include operating systems, part-picking robots, and network communication servers. In these cases, an infinite loop is an adequate model:
(`loop` $<$ *body* $>$)
The time allocation should maximize the utility gain per unit time, that is, at each iteration we choose x such that:

$$\max_{0 \le x \le t} \{Q_S(x)/x\}$$

where $Q_S(x)$ is the PP of the body of the loop. This amounts to stopping the sequence when it reaches the point of contact of the steepest tangent to the PP (figure 4). Loops with anytime termination tests offer more complex but very interesting optimization problems.

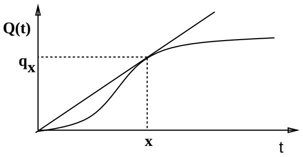

Figure 4: Compilation of a loop

4.3 Compiling interruptible algorithms

With interruptible algorithms we cannot simply allocate a certain amount of time to each component since we do not know the total computation time in advance. For example, in the case of the repair system mentioned earlier, if we allocate a certain amount of time to 'diagnose' and the execution is interrupted before 'therapy' starts, then there will be no results to report. We therefore have to interleave the execution of all the components so that results are generated continuously.

The compilation of interruptible algorithms is solved by reduction to contract algorithms using Theorem 1. The idea is to create the best contract algorithm, using the compilation methods described above, and then create an interruptible version from the contract algorithm using the iterating construction described in the proof of the theorem.

5 Conclusion

We have presented a method for constructing real-time systems based on the use of elementary anytime algorithms together with a set of compilation methods to optimally compose these algorithms. Our method is a meta-level approach in which the meta-level problem is limited to scheduling of anytime algorithms. Laffey *et al* [1988] claim that "Currently, ad hoc techniques are used for making a system produce a response within a specified time interval". Our approach has several advantages over current techniques: it achieves optimal performance not just acceptable performance; it can handle situations in which resource availability is unknown at design time; it allows for a wide range of possible run-times and hence is more flexible; it provides machine independent real-time modules. Finally, our approach avoids a time-consuming hand-tuning process associated with the construction of real-time systems because the compilation methods are mechanized.

5.1 Related work

As mentioned above, there has been considerable work on designing and using individual anytime algorithms, both before and after Dean's coining of the term 'anytime'. There has, however, been very little work capitalizing on the additional degree of freedom offered by anytime algorithms — freedom in the very general sense that the algorithm offers to fulfill an entire spectrum of input-output specifications, over the full range of run-times, rather than just a single specification. This freedom is required by a user with a time-dependent utility is running up until t; $p_t(S_i)$ is the probability that the state of the domain will be S_i at time t). Hence we get:

$$U_C(\mathcal{A}, t) = \sum_i p_t(S_i) \sum_q p_{\mathcal{A},t}(q) U([S_i, q])$$

The value of an algorithm is then defined as the maximal utility achieved by optimal time allocation, that is:

$$V(\mathcal{A}) = \max_t \{U_C(\mathcal{A}, t)\}.$$

Based on this definition we can define an order relation over anytime algorithms: we say that $\mathcal{A}_1 \succ \mathcal{A}_2$ if $V(\mathcal{A}_1) > V(\mathcal{A}_2)$. This relation replaces the traditional notion of correctness.

4 Compilation of anytime algorithms

The compilation of anytime algorithms is the process of constructing an optimal anytime algorithm using anytime algorithms as components. Creating interruptible algorithms directly is complicated, because the total time allocation is unknown in advance. We therefore start by considering only the construction of contract algorithms and then we extend the results to interruptible algorithms using Theorem 1.

The compilation methods that we describe in this section will be integrated into a programming language for anytime computation. Our goal is to develop a compiler for a language that might be in fact syntactically indistinguishable from simple LISP, but all of whose functions might in principle be anytime algorithms. We suggest LISP as the basic language since it is already widely used for AI applications, it allows the association of objects (such as PPs) with functions, and its functional nature is more suitable for the kind of composition of algorithms that we propose.

In our proposed language, the user simply specifies how the total real-time system is built by composing and sequencing simpler elements, and the compiler generates and inserts code for resource subdivision and scheduling given only the PPs of the most primitive routines. Furthermore, the flexibility of each function makes possible richer forms of composition than is normally available in programming languages; for example, a task can be solved by interleaving several solution methods until one produces the answer. The overall performance profile of the resulting system is computed by the compiler, allowing it to be used as a new building block for still more complex systems. The following in-flight aircraft monitoring system is an example of the kind of program that our compiler is eventually intended to handle:

```
(defun monitor ()
  (calibrate-instruments)
  (if (setq pl (passengers-missing))
      (mapc #'(lambda (p) (remove-from-plane
                     (hunt-for-bags p))) pl))
  (wait-for-take-off-permission)
  (loop
    (if (setq co (collision-imminent))
        (alert-pilot (plan-avoidance)))
    (if (setq ep (engine-problems))
        (alert-pilot
          (plan-repair ep (diagnose ep))))
    (if (setq cp (off-course
           (determine-current-position)))
        (any (alert-pilot cp)
             (plan-course-adjustment cp)
             (effect-course-adjustment cp)))))
```

In this program fragment, the only algorithms that are not anytime are `remove-from-plane`, `alert-pilot`, and `off-course`. All others could consume arbitrary amounts of resources, depending on the accuracy and certainty produced, and hence need to be scheduled appropriately.

4.1 Choosing the right type of PP

Earlier we defined three types of performance profiles the most informative of which was the probability distribution profile (PDP). This representation is both expensive to maintain and complicated to compile, especially in the continuous case. The simplest representation, the expected performance profile (PEP) is not suitable for our compilation scheme as explained below. We use performance interval profiles (PIP) that keep the lower and upper bounds of the quality of results. Assuming that performance is monotonically increasing, we can compute the quality bounds of a complex algorithm using the quality bounds of its components.

4.2 Compiling contract algorithms

We analyze first several cases of compilation of contract algorithms — that is, the problem is to produce a contract algorithm from anytime components (which can be either interruptible or contract algorithms). The constructs we consider are certainly not an exhaustive collection, but serve to illustrate the issues involved in building the compiler.

4.2.1 Sequences

We first consider the optimal composition of two anytime algorithms, $\mathcal{A}_1$ and $\mathcal{A}_2$, one of which feeds its output to the other. The repair system that was described earlier illustrates this situation. Let q_1 and q_2 be the performance profiles of $\mathcal{A}_1$ and $\mathcal{A}_2$, and let $U^*(q_1, q_2)$ be the *quality combination function*, that is, the function that defines how the quality of the module depends on the quality of the components. For each allocation of time, t, $\mathcal{A}_1$ gets x time units and $\mathcal{A}_2$ gets the remaining $t - x$ time units, where x is chosen to maximize $U^*(q_1(x), q_2(t-x))$. The performance profile of the compound module is therefore

$$q^*(t) = \max_{0 \le x \le t} U^*(q_1(x), q_2(t - x))$$

Similarly, in the case of n steps we get:

$$q^*(t) = \max_{\sum x_i = t} U^*(q_1(x_1), ..., q_n(x_n))$$

Figure 3 shows a problem with two alternative anytime algorithms that can solve the entire problem (for example, two different bin-packing algorithms for the same van on a single trip). In this case the quality combination function is the maximum of the two components

2.2 Elementary anytime algorithms

Elementary anytime algorithms are already widely available, contrary to popular supposition. Many existing general programming and reasoning techniques produce useful anytime algorithms: search techniques such as iterative deepening; asymptotically correct inference algorithms such as approximate query answering [Elkan, 1990; Vrbsky *et al.*, 1990], bounded cutset conditioning (see [Horvitz, 1987]), and variable precision logic [Michalski and Winston, 1986]; various greedy algorithms (see [Boddy and Dean, 1989]); iterative methods such as Newton's method; adaptive algorithms such as PAC learning algorithms [Haussler, 1990] or neural networks; Monte Carlo algorithms for simulating probabilistic models; and the use of optimal meta-level control of computation [Russell and Wefald, 1989].

2.3 Interruptible vs contract algorithms

As mentioned in section 1, we distinguish between two types of anytime algorithms. *Interruptible* algorithms are those whose run-time need not be determined at the time of activation. They can be interrupted at any time to yield results whose quality is characterized by their PP. Many of the elementary anytime algorithms mentioned above, such as iterative deepening algorithms, are interruptible. *Contract* algorithms require a specific time allocation when activated. For example, Korf's RTA* [1988] performs a depth-first or best-first search within a predetermined search horizon that is computed from the time allocation provided, and can therefore modeled as a contract algorithm. Although this algorithm can produce results for any given time allocation, if it is interrupted before the expiration of the allocation it may yield no results.

Every interruptible algorithm is trivially a contract algorithm, however the converse is not true. In general, the greater freedom of design makes it easier to construct contract algorithms than interruptible ones. The following theorem is therefore essential for the compilation of interruptible algorithms.

Theorem 1 *For any contract algorithm $\mathcal{A}$, an interruptible algorithm $\mathcal{B}$ can be constructed such that $q_{\mathcal{B}}(4t) \geq q_{\mathcal{A}}(t)$.*

Proof: Construct $\mathcal{B}$ by running $\mathcal{A}$ repeatedly with exponentially increasing time limits. Let the sequence of run-time segments be $\tau, 2\tau, ..., 2^i\tau, ...$, and assume that the code required to control this loop can be ignored. Note also that $\sum_{i=0}^{i=n-1} 2^i = 2^n - 1$. The worst case occurs when $\mathcal{B}$ is interrupted after almost $(2^n - 1)\tau$ time units, just before the last iteration terminates and the returned result is based on the previous iteration with a run-time of $2^{n-2}\tau$ time units. Since $\frac{2^n-1}{2^{n-2}} < 4$, we get the factor of 4. If we replace the multiplier of time intervals by α we get a time ratio of: $\frac{\alpha^n-1}{\alpha^{n-1}-\alpha^{n-2}}$. The lower bound of this expression is 4, for $\alpha = 2$, hence the above sequence of run-times is optimal[2].

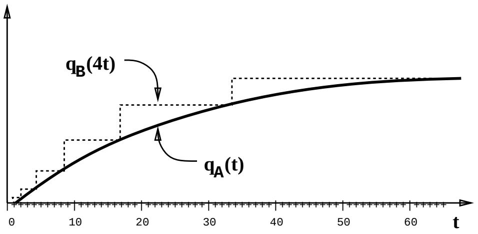

Figure 2: PP of the constructed interruptible algorithm

Note that τ may be arbitrarily small and should be in general the shortest run-time for which there is a significant improvement in the results of $\mathcal{A}$.

Figure 2 shows a typical performance profile for the contract algorithm $\mathcal{A}$, and the corresponding performance profile for the constructed interruptible algorithm $\mathcal{B}$, reduced along time axis by a factor of 4.

As an example, consider the application of this construction method to Korf's RTA*, a contract algorithm. As the time allocation is increased exponentially, the algorithm will increase its depth bound by a constant; the construction therefore generates an iterative deepening search automatically.

3 Evaluating anytime algorithms

Traditional algorithms are verified in the context of input and output predicates specified by the designer. Optimizing performance means simply reducing the execution time of a correct algorithm. The use of anytime algorithms in agents requires taking into account the real-time environment in which they operate and the utility function of the agent (or its designer). It is assumed that imprecise results have some value depending on their quality and the utility function of the system. The following framework, roughly analogous to that of [Horvitz, 1987], defines precisely what it means to be a better anytime algorithm; this depends not only on the PP of the algorithm but also on the domain and utility function (which are defined by the user).

A utility function is defined over the states of the domain: $U : \mathcal{S} \rightarrow \mathcal{R}$.

Given an algorithm $\mathcal{A}$, let $[S, q]$ denote the state of the domain that is reached by providing results of quality q when the domain is in state S. We can always view $\mathcal{A}$ as a decision making algorithm and the new state is simply the state resulting from performing the action recommended by the $\mathcal{A}$.

Now, given the current state S_0 and a certain time period t, we want to compute the comprehensive utility of the results produced by $\mathcal{A}$ with computation time t. The problem is that there is uncertainty concerning:

1. The quality of results of $\mathcal{A}$ at time t.
2. The state of the domain at time t.

The probabilistic description of the former is given by the PP of $\mathcal{A}$, and the probabilistic description of the latter is given by the model of the environment (we assume that the environment is not affected by which algorithm

[2] This factor can obviously be reduced by scheduling the contract algorithm on multiple processors. The parallel scheduling options are non-trivial, and are not discussed in this paper.

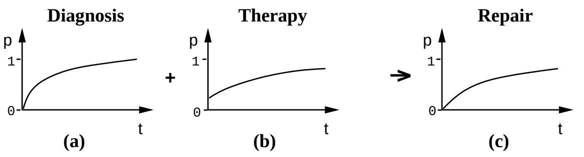

Figure 1: Composition of anytime algorithms

by a simple call-return mechanism. However, when algorithms have resource allocation as a degree of freedom, there arises the question of how to construct, for example, the optimal composition of two anytime algorithms, one of which feeds its output to the other. Consider making a repair system from a 'diagnosis' component and a 'therapy' component. The more time spent on diagnosis, the more likely the hypothesis is to be correct. The more time spent on therapy planning, the more likely the problem is to be fixed, *assuming the diagnosis is correct.* Given the performance profiles of the two subsystems (as shown in Figure 1ab), it is straightforward to construct the optimal apportionment of resources for a given total allocation, and hence to construct the optimal anytime algorithm for the whole problem (whose performance profile is shown in Figure 1c).

To summarize, in our approach the user specifies the structural decomposition of a complex problem into elementary performance components, each of which can be a traditional or an anytime algorithm. For example, the repair system might be specified as:

```
(defun repair (x)
  (apply-therapy x (diagnose x)))
```

Our system generates an anytime algorithm for the original problem by scheduling and monitoring the components in an optimal way (with respect to a given utility function). The rest of this paper describes this method in detail. In Section 2 we define the probabilistic description of the performance of anytime algorithms and examine their essential properties. In Section 3 we present a framework for evaluating anytime algorithms within the context of a given domain and a utility function. In Section 4 we explain and demonstrate the task of compiling anytime algorithms. Finally, Section 5 summarizes the benefits of our approach and discusses related work and further work to be undertaken.

2 Anytime algorithms

Anytime algorithms are characterized by their *performance profile* (PP), a probabilistic description of the quality of results as a function of time. The exact meaning and concrete representation of a PP is implementation dependent. In this section we define three types of PP and explain the basic properties of anytime algorithms.

2.1 Performance profiles

A PP maps computation time to a probabilistic description of the quality of the results. The main reason for the uncertainty concerning the quality of results (especially with deterministic algorithms) is the fact that the input to the algorithm is unknown. Therefore, a PP should always be interpreted with respect to a particular probability distribution of input.

Given an anytime algorithm $\mathcal{A}$, let $q_{\mathcal{A}}(x,t)$ be the quality of results produced by $\mathcal{A}$ with input x and computation time t; let $q_{\mathcal{A}}(t)$ be the expected quality of results with computation time t; and let $p_{\mathcal{A},t}(q)$ be the probability (density function in the continuous case) that $\mathcal{A}$ with computation time t produces results of quality q. A complete description of the performance of $\mathcal{A}$ is given by the following definition:

Definition 2.1 *The* **performance distribution profile (PDP)***, of an algorithm $\mathcal{A}$ is a function $D_{\mathcal{A}} : \mathcal{R}^+ \rightarrow Pr(\mathcal{R})$ that maps computation time to a probability distribution of the quality of the results.*

In some cases the summation over all possible inputs may produce too wide a range of qualities and the information provided by the PP may be too general. In that case we use conditional PPs by partitioning the input domain into classes and storing a separate PP for each class (this partitioning is done using any attribute of the input, such as size or a complexity measure).

Definition 2.2 *The* **expected performance profile (PEP)***, of an algorithm $\mathcal{A}$ is a function $E_{\mathcal{A}} : \mathcal{R}^+ \rightarrow \mathcal{R}$ that maps computation time to the expected quality of the results.*

Note that $E_{\mathcal{A}}(t) = \sum_q p_{\mathcal{A},t}(q)q = \sum_x Pr(x)q_{\mathcal{A}}(x,t)$. This is exactly what Dean and Boddy [1988] used as a performance profile.

Definition 2.3 *The* **performance interval profile (PIP)***, of an algorithm $\mathcal{A}$ is a function $I_{\mathcal{A}} : \mathcal{R}^+ \rightarrow \mathcal{R} \times \mathcal{R}$ that maps computation time to the upper and lower bounds of the quality of the results.*

Note that if $I_{\mathcal{A}}(t) = [L, U]$ then $\forall x : L \leq q_{\mathcal{A}}(x,t) \leq U$.

The quality of results described by a PP is measured in one of the following ways:

1. **Certainty** – Probability of correctness determines quality (e.g. randomized algorithms for primality testing).
2. **Accuracy** – Error bound determines quality (e.g. Newton's method).
3. **Specificity** – Amount of detail determines quality (e.g. hierarchical diagnosis).

While accuracy is typically used to measure quality in numerical domains, and specificity in symbolic domains, the former can be seen as a special case of the latter; an inaccurate numerical solution is very specific but incorrect, and could be mapped to an equally useful, correct statement that the solution lies within a certain interval. Anytime algorithms can also have multidimensional quality measures, for example PAC algorithms for inductive learning are characterized by an uncertainty measure δ and a precision measure ϵ.

**To appear in Proceedings of the 12th International Joint Conference on Artificial Intelligence
August 24-30, 1991, Sydney, Australia**

Composing Real-Time Systems

Stuart J. Russell and **Shlomo Zilberstein**
Computer Science Division
University of California
Berkeley, California 94720 U.S.A.
russell@colditz.berkeley.edu shlomo@bastille.berkeley.edu

Abstract

We present a method to construct real-time systems using as components *anytime* algorithms whose quality of results degrades gracefully as computation time decreases. Introducing computation time as a degree of freedom defines a scheduling problem involving the activation and interruption of the anytime components. This scheduling problem is especially complicated when trying to construct *interruptible* algorithms, whose total run-time is unknown in advance. We introduce a framework to measure the performance of anytime algorithms and solve the problem of constructing interruptible algorithms by a mathematical reduction to the problem of constructing *contract* algorithms, which require the determination of the total run-time when activated. We show how the composition of anytime algorithms can be mechanized as part of a compiler for a LISP-like programming language for real-time systems. The result is a new approach to the construction of complex real-time systems that separates the arrangement of the performance components from the optimization of their scheduling, and automates the latter task.

1 Introduction

Our objective in this research has been to develop and automate a methodology for the construction of utility-driven, real-time agents. A real-time agent is an agent whose utility function depends on time. For example, a utility function defined as the number of widgets assembled per hour depends on time; a robot designed to maximize this utility function is a real-time agent. Similarly, problems such as chess-playing, reentry navigation for a space shuttle, financial planning and trading, and medical monitoring in an intensive care unit have utility functions that depend on time, and therefore require the construction of real-time systems. This approach generalizes the traditional view of real-time systems as systems that can guarantee a response after a fixed time has elapsed [Laffey *et al.*, 1988], in that deadlines can be expressed by a sharp drop in the utility function.

We show in this paper how to construct real-time systems using anytime algorithms[1] as basic blocks. Anytime algorithms are algorithms whose quality of results degrades gracefully as computation time decreases, hence they introduce a tradeoff between computation time and quality of results. The algorithm's *performance profile* (PP) gives a probabilistic description of the quality of results as a function of time (we define and generalize this notion in section 2). For example, consider a hierarchical diagnosis algorithm that recursively performs a test to identify the defective component of an assembly. This algorithm can be interrupted at any time to produce a partial diagnosis whose quality can be measured by the level of specificity. By translating the quality of results into a utility measure that takes into account the time needed to produce these results, we can compute the optimal amount of time that should be allocated to diagnosis, after which a complete defective component should be replaced rather than being further analyzed. A similar technique was used by Boddy and Dean [1989] for solving a real-time path planning problem and by Horvitz [1987] for real-time decision making in the health care domain.

An important distinction that has to some extent been ignored in the literature should be made between *interruptible* algorithms and *contract* algorithms. Interruptible algorithms produce results of the 'advertised quality' even when interrupted unexpectedly; whereas contract algorithms, although capable of producing results whose quality varies with time allocation, must be given a particular time allocation in advance. If a contract algorithm is interrupted at any time shorter than the contract time it may yield no useful results. An important result of this paper, given in section 2, shows that there is in fact a simple reduction from interruptible algorithms to contract algorithms.

In this work we extend the use of anytime algorithms to the construction of complex real-time systems. It is unlikely that a complex system will be developed by implementing one large anytime algorithm. Systems are normally built from components that are developed and tested separately. In standard algorithms, the quality of the output is fixed, so composition can be implemented

[1] Dean and Boddy [1988] coined the term "anytime algorithm" in their paper on time-dependent planning.