AIAA-94-1292-CP

REAL-TIME ROBOT DELIBERATION BY COMPILATION AND MONITORING OF ANYTIME ALGORITHMS

Shlomo Zilberstein
Computer Science Department
University of Massachusetts at Amherst

Abstract

This paper addresses a central issue in robot construction, namely the control of deliberation time. The complexity of automated planning and scheduling makes it undesirable, sometimes infeasible, to find the optimal action in every situation since the deliberation process itself degrades the performance of the system. The question is how can an intelligent robot react to a situation after performing the "right" amount of thinking. It is by now widely accepted that a successful robotic system must trade off between decision quality and the computational resources used to produce it. Anytime algorithms, introduced by Dean, Horvitz and others in the late 1980's, were designed to offer such a trade-off. Recent work by Zilberstein and Russell shows that the advantages of anytime algorithms can be extended to the construction of complex robotic systems. This paper describes the compilation and monitoring mechanisms that are required to build robots that can efficiently control their deliberation time.

I. Control of Deliberation Time

Intelligent robots must perform real-time deliberation to solve such problems as path planning, task scheduling, and interpretation of sensory data. An important aspect of intelligent behavior is the capability of robots to factor the cost of deliberation into the deliberation process. Two factors determine the cost of deliberation: the resources consumed by the process, primarily computation time, and constant change in the environment that may decrease the relevance of the outcome and hence reduce its value. A useful mechanism to quantify this dependency is based on the definition of a utility function $U(S)$ over the states of the world. The utility of a state defines the desirability of that state. For example, the utility of a robot that assembles a certain product can be measured by the number of products completed each hour. Utility functions extend the traditional notion of deadline (allowing for gradual decrease of value over time) and the traditional notion of goals (allowing for partial goal satisfaction). Thus, they are more suitable for control of real-time robotic systems.

To choose an optimal course of action and to maximize its utility function, a robot must perform some real-time problem solving. The performance of robotic systems can be improved by optimizing the quality of their decisions *net* of deliberation cost. The problem of deliberation cost has been widely discussed in economics, engineering and artificial intelligence. In artificial intelligence, researchers have proposed a number of meta-level architectures to control the cost of base-level reasoning [3, 9, 14]. The model presented in this paper belongs to this class of solutions: its meta-level reasoning component optimizes resource allocation to the base-level performance components. This approach separates two, central aspects of robot construction: the development of the performance components and the optimization of performance. This modularity is accomplish by using anytime algorithms as the elementary components of the system.

The rest of the paper describes our approach in detail. Section II describes the notion of anytime algorithms. It shows how to construct anytime algorithms and how to characterize the trade-off that they offer between quality of results and computation time. Section III explains the benefits and difficulties involved in the composition of anytime algorithms. Sections IV and V describe the two main components of our solution to the composition problem, namely off-line compilation and run-time monitoring. Section VI describes briefly some applications of this approach. Finally, Section VII summarizes the benefits of our approach and discusses some directions for further work.

II. Anytime Algorithms

The term "anytime algorithm" was coined by Dean in the late 1980's in the context of his work on time-dependent planning. Anytime algorithms are algorithms whose quality of results improves gradually as computation time increases, hence they offer a tradeoff between resource consumption and output quality. Many numerical approximation methods, such as Taylor series approximation, are based on iterative improvement and, as such, can be considered an anytime algorithm.

Various metrics can be used to measure the quality of a result produced by an anytime algorithm. From a pragmatic point of view, it may seem useful to define a *single* type of quality measure to be applied to all anytime algorithms. Such a unifying approach may simplify the meta-level control. However, in practice, different types of anytime algorithms tend to approach the exact result in completely different ways. The following metrics have been proved useful in anytime algorithm construction:

1. **Certainty** – this metric reflects the degree of certainty that the result is correct. The degree of certainty can be expressed using probabilities, fuzzy set membership, or any other approach.
2. **Accuracy** – this metric reflects the degree of accuracy or how close is the approximate result to the exact answer. Normally with such algorithms, high quality provides a *guarantee* that the error is below a certain small upper bound.
3. **Specificity** – this metric reflects the level of detail of the result. In this case, the anytime algorithm always produces *correct* results, but the level of *detail* is increased over time.

Many existing programming techniques produce useful anytime algorithms. Examples include iterative deepening search, variable precision logic, and randomized techniques such as Monte Carlo algorithms or fingerprinting algorithms. For a survey of such programming techniques and examples of algorithms see [21].

The notion of interrupted computation is almost as old as computation itself. However, traditionally, interruption was used primarily for two purposes: aborting the execution of an algorithm whose results are no longer necessary, or suspending the execution of an algorithm for a short time because a computation of higher priority must be performed. Anytime algorithms offer a third type of interruption: interruption of the execution of an algorithm whose results are considered "good enough" by their consumer.

Conditional performance profiles

To allow for efficient meta-level control of anytime algorithms, we characterize their behavior by *conditional performance profiles* (CPP) [19]. A conditional performance profile captures the dependency of output quality on time allocation as well as on input quality. In [21], the reader can find a detailed discussion of various types of conditional performance profiles and their representation. To simplify the discussion of compilation, we will refer only to the *expected* CPP that maps computation time and input quality to the expect output quality.

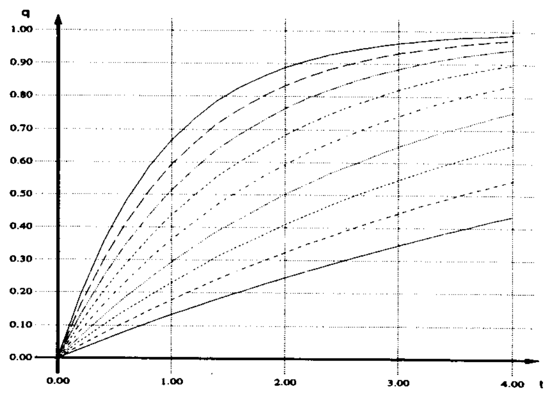

Figure 1: Graphical representation of a CPP

Definition 1 *The conditional performance profile (CPP), of an algorithm $\mathcal{A}$ is a function $CPP_{\mathcal{A}} : \mathcal{Q}_{in} \times \mathcal{R}^+ \rightarrow \mathcal{Q}_{out}$ that maps input quality and computation time to the expected quality of the results.*

Figure 1 shows a typical CPP. Each curve represents the expected output quality as a function of time for a *given* input quality.

Interruptible and contract algorithms

In [15] we make an important distinction between two types of anytime algorithms, namely interruptible and contract algorithms. An interruptible algorithm can be interrupted at any time to produce results whose quality is described by its performance profile. A contract algorithm offers a similar trade-off between computation time and quality of results, but it must know the total allocation of time in advance. If interrupted at any point before the termination of the contract time, it may yield no useful results. Interruptible algorithms are in many cases more appropriate for the application, but they are also more complicated to construct. In [15] we show that a simple, general construction can produce an interruptible version for any given contract algorithm, with only a small, constant penalty. This theorem allows us to concentrate on the construction of contract algorithms for complex decision-making tasks and then convert them into interruptible algorithms using a standard transformation.

III. Composing Anytime Algorithms

Modularity is widely recognized as an important issue in system design and implementation. However, the use of anytime algorithms as the components of a modular system presents a special type of scheduling problem. The question is how much time to allocate to each component in order to maximize the output quality of the complete system. We

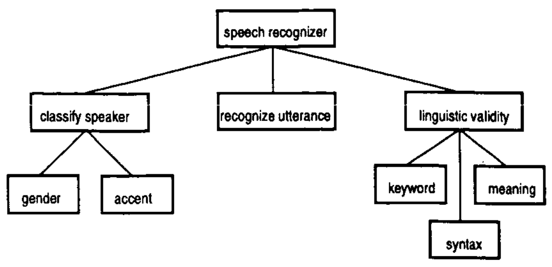

Figure 2: A composite module for speech recognition

refer to this problem as the anytime algorithm *composition problem.*

Consider for example a speech recognition system whose structure is shown in Figure 2. Each box represents an elementary anytime algorithm whose conditional performance profile is given. The system is composed of three main components. First, the speaker is classified in terms of gender and accent. Then a recognition algorithm suggests several possible matching utterances. And finally, the linguistic validity of each possible utterance is determined and the best interpretation is selected. The composition problem is the problem of calculating how much time to allocate to each elementary component of the composite system, so as to maximize the quality of the utterance recognition.

Solving the composition problem is important for several reasons. First, it introduces a new kind of modularity into real-time system development by allowing for separation between the development of the performance components and the optimization of their performance. In traditional design of real-time systems, the performance components must meet certain time constraints that are not always known at design time. The result is a hand-tuning process that, may or may not, culminate with a working system. Anytime computation offers an alternative to this approach. By developing performance components that are responsive to a wide range of time allocations, one avoids the commitment to a particular performance level that might fail the system.

The second reason why the composition problem is important relates to the difficulty of programming with anytime algorithms. To make a composite system optimal (or even executable), one must control the activation and interruption of the components. In solving the composition problem, our goal is to minimize the responsibility of the programmer regarding this optimization problem. Our solution is described in the following two sections.

IV. Compilation

Given a system composed of anytime algorithms, the compilation process is designed to: (a) determine the optimal performance profile of the complete system; and (b) insert into the composite module the necessary code to achieve that performance. The precise definition and solution of the problem depend on the following factors:

1. **Composite program structure** – what type of programming operators are used to compose anytime algorithms?
2. **Type of performance profiles** – what kind of performance profiles are used to characterize elementary anytime algorithms?
3. **Type of anytime algorithms** – what type of elementary anytime algorithms are used as input? what type of anytime algorithm should the resulting system be?
4. **Type of monitoring** – what type of run-time monitoring is used to activate and interrupt the execution of the elementary components?
5. **Quality of intermediate results** – what access does the monitoring component have to intermediate results? is the actual quality of an intermediate result known to the monitor?

Depending on these factors, different types of compilation and monitoring strategies are needed. To simplify the discussion in this paper, we will consider only the problem of producing contract algorithms when the conditional performance profiles of the components are given. We will assume that no active monitoring is allowed once the system is activated. A broader, in-depth analysis of compilation and monitoring can be found in [21].

Let $\mathcal{F}$ be a set of anytime functions. Assume that all function parameters are passed by value and that functions have no side-effects (as in pure functional programming). Let $\mathcal{I}$ be a set of input variables. Then, the notion of a composite expression is defined as follows:

Definition 2 *A **composite expression** over $\mathcal{F}$ with input $\mathcal{I}$ is:*

1. *An expression $f(i_1, ..., i_n)$ where $f \in \mathcal{F}$ is a function of n arguments and $i_1, ..., i_n \in \mathcal{I}$.*
2. *An expression $f(g_1, ..., g_n)$ where $f \in \mathcal{F}$ is a function of n arguments and each g_i is a composite expression or an input variable.*

For example, the expression $A(B(x), C(D(y)))$ is a composite expression over $\{A, B, C, D\}$ with input $\{x, y\}$. Suppose that each function in $\mathcal{F}$ has a conditional performance profile associated with it that specifies the quality of its output as a function of time allocation to that function and the qualities of its inputs. Given a composite expression of size n, the main part of the compilation process is to determine a mapping:

$$\mathcal{T} : t \rightarrow (t_1, ..., t_n) \qquad (1)$$

This mapping determines for each total allocation, t, the allocation to the components that maximizes the output quality.

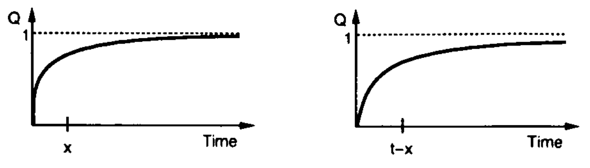

Figure 3: The performance profiles of $\mathcal{A}_1$ and $\mathcal{A}_2$

A compilation example

Let us look first at a simple example of compilation involving only two anytime algorithms. Suppose that one algorithm takes the input and produces an intermediate result. This result is then used as input to another anytime algorithm which, in turn, produces the final result. Many systems can be implemented by a composition of a sequence of two or more algorithms. For example, an automated repair system can be composed of two algorithms: diagnosis and treatment. This can be represented in general by the following expression:

$$Output \leftarrow \mathcal{A}_2(\mathcal{A}_1(Input))$$

Figure 3 shows the performance profiles of $\mathcal{A}_1$ and $\mathcal{A}_2$. These performance profiles are defined by:

$$Q_1(t) = 1 - e^{-\lambda_1 t} \qquad Q_2(t) = 1 - e^{-\lambda_2 t}$$

Assume that the output quality is the sum of the qualities of $\mathcal{A}_1$ and $\mathcal{A}_2$, then the following result holds:

Theorem 3 *Given the performance profiles of $\mathcal{A}_1$ and $\mathcal{A}_2$, the optimal time allocation mapping is:*

$$\mathcal{T} : t \rightarrow \left(\frac{\ln \lambda_1 - \ln \lambda_2 + \lambda_2 t}{\lambda_1 + \lambda_2}, \frac{\ln \lambda_2 - \ln \lambda_1 + \lambda_1 t}{\lambda_1 + \lambda_2}\right) \quad (2)$$

Proof: Since the overall output quality is:

$$Q(x) = 1 - e^{-\lambda_1 x} + 1 - e^{-\lambda_2 (t-x)} \quad (3)$$

the maximal quality is achieved when $\frac{\partial Q}{\partial x} = 0$.
In other words:

$$\lambda_1 e^{-\lambda_1 x} - \lambda_2 e^{-\lambda_2 (t-x)} = 0 \quad (4)$$

The solution of this equation yields the above allocation. □

To complete the compilation process, the compiler needs to insert code in the original expression for proper activation of $\mathcal{A}_1$ and $\mathcal{A}_2$ as contract algorithms with the appropriate time allocation. This is done by replacing the simple function call by an anytime function call [21]. The implementation of an anytime function call depends on the particular programming environment and will not be discussed in this paper.

The complexity of compilation

The compilation problem is defined as an optimization problem, that is, a problem of finding a schedule of a set of components that yields maximal output quality. In order to analyze its complexity, it is more convenient to refer to the decision problem variant of the compilation problem. Given a composite expression e, the conditional performance profiles of its components, and a total allocation B, the decision problem is whether there exists a schedule of the components that yields output quality greater than or equal to K. To begin, consider the general problem of global compilation of composite expressions, or GCCE. In [21], we prove the following result:

Theorem 4 *The GCCE problem is NP-complete in the strong sense.*

The proof is based on a reduction from the PARTIALLY ORDERED KNAPSACK problem which is known to be NP-complete in the strong sense. The meaning of this result is that the application of the compilation technique may be limited to small programs. To address the complexity problem of global compilation, we developed an efficient local compilation technique.

Local compilation

Local compilation is the process of finding the best performance profile of a module based on the performance profiles of its *immediate* components. If those components are not elementary anytime algorithms, then their performance profiles are determined using local compilation. Local compilation replaces the global optimization problem with a set of simpler, local optimization problems and reduce the complexity of the whole problem. Unfortunately, local compilation cannot be applied to every composite expression. If the expression has repeated subexpressions, then computation time should be allocated only once to evaluate all identical copies. Local compilation cannot handle such cases. However, the following three assumptions make local compilation both efficient and optimal [21]:

1. **The tree-structured assumption** – the input composite expression has no repeated subexpressions, thus its DAG (directed acyclic graph) representation is a tree.
2. **The input-monotonicity assumption** – the output quality of each module increases when the quality of the input improves.
3. **The bounded-degree assumption** – the number of inputs to each module is bounded by a constant, b.

Under these assumptions, local compilation is both efficient and yields optimal results [21]. The first assumption is needed so that local compilation can be applied. The second assumption is needed to guarantee the optimality of the resulting performance profile. And the third assumption is needed to guarantee the efficiency of local compilation. Using an efficient tabular representation of performance profiles, we could perform local compilation in constant

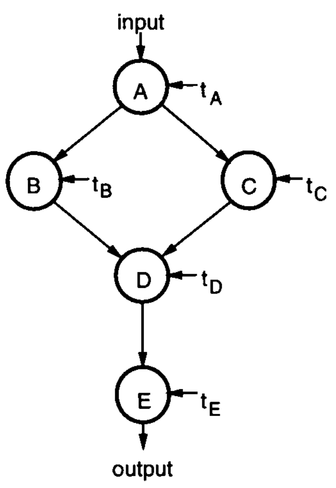

Figure 4: DAG representation of F

time and reduce the overall complexity of compilation to be linear in the size of the program.

Repeated subexpressions

While the input-monotonicity and the bounded-degree assumptions are quite reasonable (and also desirable from a methodological point of view), the tree-structured assumption is somewhat restrictive. We want to be able to handle the case of repeated subexpressions. To understand the problem, consider the following expression:

$$F = E(D(B(A(x)), C(A(x))))$$

Figure 4 shows the DAG representation of F. Recall that the purpose of compilation is to compute a time allocation mapping that would specify for each input quality and total allocation of time the best apportionment of time to the components so as to maximize the expected quality of the output. But local compilation is only possible when one can repeatedly break a program into sub-programs whose execution intervals are disjoint, so that allocating a certain amount of time to one sub-program does not affect in any way the evaluation and quality of the other sub-programs. This property does not hold for DAGs. In the example shown in Figure 4, B and C are the ancestors of D, but their time allocations cannot be considered independently since they both use the same sub-expression, $A(x)$.

To address this problem we have developed a number of *approximate* compilation techniques that work efficiently on DAGs, but do not guarantee optimality of the schedule [21]. The compilation of additional programming constructs, such as conditional statements and loops, is analyzed in [21]. To summarize, a number of compilation techniques have been developed that can efficiently produce the performance profile of a composite system based on the performance profiles of its components.

V. Run-Time Monitoring

Monitoring plays a central role in anytime computation as it complements anytime algorithms with a mechanism that determines their run-time. We examine the monitoring problem in two types of domains. One type is characterized by the predictability of utility change over time. High predictability of utility allows an efficient use of contract algorithms modified by various strategies for contract adjustment. The second type of domains is characterized by rapid change and a high level of uncertainty. In such domains, active monitoring, that schedules interruptible algorithms based on the value of computation criterion, becomes essential.

Given a compound anytime program, $\mathcal{P}$, whose elementary anytime components are $E = \{\mathcal{A}_1, ..., \mathcal{A}_n\}$, a monitoring scheme is defined as a mapping that determines a certain time allocation for each activation of an elementary component.

Definition 5 *A **monitoring scheme** for a program $\mathcal{P}$ is a mapping:*

$$\mathcal{M} : E \times Z^+ \rightarrow R^+$$

where E is the set of elementary components of $\mathcal{P}$.

$\mathcal{M}(i, j)$ is the time allocation to the j^{th} activation of the i^{th} component. A monitoring scheme supplies the necessary information to make a compound anytime program executable in a well defined way. In defining monitoring schemes, we make a distinction between *passive* and *active* monitoring.

Definition 6 *A monitoring scheme is said to be **passive** if the corresponding time allocation mapping is completely determined prior to the activation of the system.*

Definition 7 *A monitoring scheme is said to be **active** if it is not passive. That is, the corresponding time allocation mapping is partially determined while the system is active.*

Under active monitoring, some scheduling decisions are made at run-time. Such decisions are based on the *actual* quality of results produced by the anytime components and based on the *actual* change that occurred in the environment. The main reason why active monitoring is necessary in control of anytime algorithms is the problem of uncertainty. In an entirely deterministic world, passive monitoring can yield optimal performance. However, in unpredictable domains there is much to be gained in performance by introducing an active monitoring component.

Two primary sources of uncertainty affect the operation of real-time robotic systems. The first source is internal to the system. It is caused by the unpredictable behavior of the system itself. The second source is external. It is caused by unpredictable changes in the environment. These two sources of uncertainty are characterized by two separate knowledge sources. Uncertainty regarding the performance of the system is characterized by the performance profile of the system (in particular, we use *performance distribution profiles* to represent the probability distribution of quality of results). Uncertainty regarding the future state of the environment is characterized by the model of the environment. Obviously, the type of active monitoring may vary as a function of the source of uncertainty and the degree of uncertainty.

Monitoring contract algorithms

It is easier to construct contract algorithms than interruptible ones, both as elementary and as compound algorithms. Therefore, I will examine first the monitoring problem assuming that the complete system is presented as a contract algorithm, $\mathcal{A}$. The conditional performance profile of the system is $Q_{\mathcal{A}}(q,t)$ where q is the input quality and t is the time allocation. Assume that $Q_{\mathcal{A}}(q,t)$ represents, in the general case, a probability distribution. When a discrete representation is used, $Q_{\mathcal{A}}(q,t)[q_i]$ denotes the probability of output quality q_i.

Let S_0 be the current state of the domain and let S_t represent the state of the domain at time t, let q_t represent the quality of the result of the contract anytime algorithm at time t. $U_{\mathcal{A}}(S,t,q)$ represents the utility of a result of quality q in state S at time t. This utility function is given as part of the problem description. The purpose of the monitor is to maximize the expected utility of the result, that is, to find t for which $U_{\mathcal{A}}(S_t,t,q_t)$ is maximal. Contract algorithms are especially useful in a particular type of domains which is defined as follows:

Definition 8 *A domain is said to have* ***predictable utility*** *if $U_{\mathcal{A}}(S_t,t,q)$ can be determined for any future time, t, and quality of results, q, once the current state of the domain, S_0, is known.*

The notion of predictable utility is a property of domains. The same utility function can be predictable in one domain and unpredictable in another. What makes a domain predictable is the capability to determine the exact value of results of a particular quality at any future time. Hence, the state of the domain may change, even in an unpredictable way, and utility may still be predictable. To explain this situation, we define a function, $f(S)$, that isolates the features of a state that determine its utility. In other words,

$$\forall S_1,S_2\ f(S_1)=f(S_2)\ \Rightarrow\ U_{\mathcal{A}}(S_1,t,q)=U_{\mathcal{A}}(S_2,t,q) \tag{5}$$

Consider for example a transportation domain that refers to traffic on a particular road. The state of the domain is defined by the location and velocity of each vehicle and $f(S)$ may be, for example, the traffic density. Using the function f, it is easy to show that a domain with predictable utility is a domain for which $f(S_t)$ can be determined once the current state, S_0, is known. In general, three typical cases of such domains can be identified:

1. A static domain is obviously predictable since $S_t = S_0$ and $f(S_t)=f(S_0)$. For example, the game of chess constitutes a static domain.
2. A domain that has a deterministic model is predictable since future states can be uniquely determined and hence $f(S_t)$ can be determined. For example, a domain that includes moving objects has a deterministic model when the velocity of each object is constant.
3. A domain for which there is a deterministic model to compute $f(S_t)$, once the current state is known, is predictable. Note that this does not require a deterministic model of the domain itself. An important sub-class is all the domains for which $f(S)=\emptyset$, that is, domains in which the utility function depends only on time.

The initial contract time

The first step in monitoring contract algorithms involves the calculation of the initial contract time. Due to uncertainty concerning the quality of the result of the algorithm, the expected utility of the result at time t is represented by:

$$U'_{\mathcal{A}}(S_t,t)=\sum_i Q_{\mathcal{A}}(q,t)[q_i]U_{\mathcal{A}}(S_t,t,q_i) \tag{6}$$

The probability distribution of future output quality is provided by the performance profile of the algorithm. Hence, an initial contract time, t_c, can be determined before the system is activated by solving the following equation:

$$t_c = arg\max_t\{U'_{\mathcal{A}}(S_t,t)\} \tag{7}$$

Under passive monitoring, this initial contract time is used to determine (using the compiled performance profile of the system) the ultimate allocation to each component.

In some cases, it is possible to separate the value of the results from the time used to generate them. In such cases, one can express the comprehensive utility function, $U_{\mathcal{A}}(S,t,q)$ as the difference between two functions:

$$U_{\mathcal{A}}(S_t,t,q)=V_{\mathcal{A}}(S_0,q)-Cost(S_0,t) \tag{8}$$

where $V_{\mathcal{A}}(S,q)$ is the value of a result of quality q in a particular state S (termed *intrinsic utility* [14]) and $Cost(S,t)$ is the cost of t time units provided that the current state is S. Similar to the expected utility, the expected intrinsic utility for any allocation of time can be calculated using the performance profile of the algorithm:

$$V'_{\mathcal{A}}(S,t)=\sum_i Q_{\mathcal{A}}(q,t)[q_i]V_{\mathcal{A}}(S,q_i) \tag{9}$$

Finally, the initial contract time can be determined by solving the following equation:

$$t_c = arg\max_t\{V'_{\mathcal{A}}(S_0,t)-Cost(S_0,t)\} \tag{10}$$

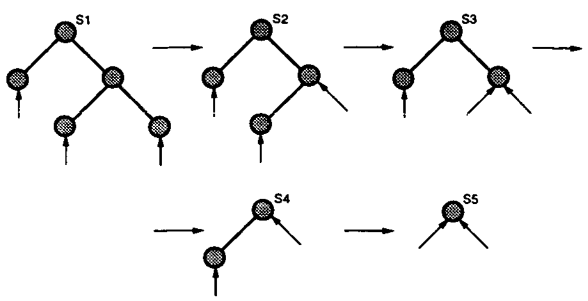

Figure 5: A sequence of residual sub-systems

Once an initial contract time is determined, several monitoring policies can be applied. The most trivial one is the *fixed-contract* strategy that leads to a passive monitoring scheme. Under this strategy, the initial contract time and the compiled performance profile of the system are used to determine the allocation to the components. This allocation remains constant until the termination of the problem solving episode. The fixed-contract policy is optimal under the following conditions:

Theorem 9 ***Optimality of monitoring of contract algorithms.*** *The fixed-contract monitoring strategy is optimal when the domain has predictable utility and the system has a fixed performance profile.*

Proof: This result is rather trivial since, when the domain has predictable utility and the system's performance profile is fixed, utility of results at any future time can be determined. The initial contract time, that maximizes the comprehensive utility, remains the same during the computation and no additional scheduling decision can improve the performance of the system. □.

We now look at two extensions to the fixed-contract policy for cases with high degree of uncertainty regarding the quality of the results. In such cases, the initial contract time must be altered by an active monitoring component.

Re-allocating residual time

The first type of active monitoring that we analyze involves reallocation of residual time among the remaining anytime algorithms. Suppose that a system, composed of several elementary contract algorithms, is compiled into an optimal compound contract algorithm. Since the results of the elementary contract algorithms are not available during their execution, the only point of time where active monitoring can take place is *between* activations of the elementary components. Based on the structure of the system, an execution order can be defined for the elementary components. The execution of any elementary component can be viewed as a transformation of a node in the graph representing the program from a computational node to an external "input" of a certain quality. This transformation is shown in Figure 5. The quality of the new input is only known when the corresponding elementary component terminates. Based on the actual quality, the remaining time (with respect to the global contract) can be reallocated among the remaining computational components to yield a performance improvement with respect to allocation that was based on the probabilistic knowledge of quality of intermediate results.

In order to be able to allocate time optimally to each component, the monitor needs to access not only the performance profile of the complete system, but also the performance profiles of the residual sub-systems. The compilation problem has to be solved for each residual system. For example, for the system modeled by Figure 5, five performance profiles must be calculated. These performance profiles can be derived using the standard local compilation technique. The only difference is that the compiler does not need to store the allocation to all the components but only the allocation to the next component in the activation order.

Adjusting contract time

The second type of active monitoring for contract algorithms involves adjustments to the original contract time. As before, once an elementary component terminates, the monitor can consider its output as an input to a smaller residual system composed of the remaining anytime algorithms. By solving the previous equation that determines the contract time for the residual system, a better contract time can be determined that takes into account the actual quality of the intermediate results generated so far.

If the elementary components are interruptible, the contract time can be adjusted *while* an elementary component is running. Given the quality of the results generated by that component and its performance profile, a new contract may be determined. In that case, the new contract may affect the termination time of the currently active module in addition to affecting the run-time of future modules.

Monitoring interruptible algorithms

We turn now to the problem of monitoring interruptible anytime computation. The use of interruptible algorithms is necessary in domains whose utility function is not predictable (and cannot be approximated by a predictable utility function). Such domains are characterized by nondeterministic rapid change. Medical diagnosis in an intensive care unit, trading in the stock exchange market, and vehicle control on a highway are examples of such domains. Many possible events can change the state of such domains and the timing of their occurrence is essentially unpredictable. Consequently, accurate projection into the far future is very limited and the previous fixed-contract approach fails. Such domains require interruptible decision making.

Active monitoring using the value of computation

Consider a system whose main decision component is an interruptible anytime algorithm, $\mathcal{A}$. The conditional probabilistic performance profile of the algorithm is $Q_{\mathcal{A}}(q,t)$ where q is the input quality and t is the time allocation. As before, $Q_{\mathcal{A}}(q,t)$ is a probability distribution and $Q_{\mathcal{A}}(q,t)[q_i]$ denotes the probability of output quality q_i.

Let S be the current state of the domain. Let S_t be the state of the domain at time t. And, let q_t represent the quality of the result of the interruptible anytime algorithm at time t. $U_{\mathcal{A}}(S,t,q)$ represents the utility of a result of quality q in state S at time t. The purpose of the monitor is to maximize the expected utility by interrupting the main decision procedure at the "right" time. Due to the high level of uncertainty in rapidly changing domains, the monitor must constantly assess the value of continued computation by calculating the net expected gain from continued computation given the current best results and the current state of the domain. This is done in the following way:

Due to the uncertainty concerning the quality of the result of the algorithm, the expected utility of the result in a given future state S_t at some future time t is represented by:

$$U'_{\mathcal{A}}(S_t,t) = \sum_i Q_{\mathcal{A}}(q,t)[q_i]U_{\mathcal{A}}(S_t,t,q_i) \qquad (11)$$

The probability distribution of future output quality is provided by the performance profile of the algorithm. Due to the uncertainty concerning the future state of the domain, the expected utility of the results at some future time t is represented by:

$$U''_{\mathcal{A}}(t) = \sum_S p(S_t = S)U'_{\mathcal{A}}(S,t) \qquad (12)$$

The probability distribution of the future state of the domain is provided by the model of the environment.

Finally, the condition for continuing the computation at time t for an additional Δt time units is therefore $VOC > 0$ where:

$$VOC = U''_{\mathcal{A}}(t+\Delta t) - U''_{\mathcal{A}}(t) \qquad (13)$$

Similar to the case of contract algorithms, monitoring of interruptible systems can be simplified when it is possible to separate the value of the results from the time used to generate them. In such cases, one can express the comprehensive utility function, $U_{\mathcal{A}}(S,t,q)$, as the difference between two functions:

$$U_{\mathcal{A}}(S_t,t,q) = V_{\mathcal{A}}(S,q) - Cost([t_c,t]) \qquad (14)$$

where $V_{\mathcal{A}}(S,q)$ is the intrinsic utility function, S is the current state, t_c is the current time, and $Cost([t_c,t])$ is the cost of the time interval $[t_c,t]$. Under this separability assumption, the intrinsic value of allocating a certain amount of time t to the interruptible system (resulting in domain state S) is:

$$V'_{\mathcal{A}}(S,t) = \sum_i Q_{\mathcal{A}}(q,t)[q_i]V_{\mathcal{A}}(S,q_i) \qquad (15)$$

Hence, the intrinsic value of allocating a certain time t in the current state is:

$$V''_{\mathcal{A}}(t) = \sum_S p(S_t = S)V'_{\mathcal{A}}(S,t) \qquad (16)$$

And the condition for continuing the computation at time t for an additional Δt time units is again $VOC > 0$ where:

$$VOC = V''_{\mathcal{A}}(t+\Delta t) - V''_{\mathcal{A}}(t) - Cost([t,t+\Delta t]) \qquad (17)$$

Theorem 10 ***Optimality of monitoring of interruptible algorithms.*** *Monitoring interruptible algorithms using the value of computation criterion is optimal when $\Delta t \to 0$ and when the intrinsic value function is monotonically increasing and concave down and the time cost function is monotonically increasing and concave up.*

Proof: A function q is called *concave up* on a given interval I if it is continuous, piecewise differentiable, and $\forall x,y \in I$ for which $q'(x)$ and $q'(y)$ exist, $(x < y) \Rightarrow (q'(x) \leq q'(y))$. It is called *concave down* if $\forall x,y \in I$ for which $q'(x)$ and $q'(y)$ exist, $(x < y) \Rightarrow (q'(x) \geq q'(y))$. Note that the assumption of monotonically increasing and concave down intrinsic value function is identical to the assumption of Dean and Wellman (See [4], Chapter 8, page 364) that performance profiles have the property of *diminishing returns*.

Now, suppose that the current time is t_1 and that

$$VOC = V''_{\mathcal{A}}(t_1+\Delta t) - V''_{\mathcal{A}}(t_1) - Cost([t_1,t_1+\Delta t]) \leq 0 \qquad (18)$$

Since the intrinsic value function is concave down, it is guaranteed that for any future time $t_2 > t_1$:

$$V''_{\mathcal{A}}(t_2+\Delta t) - V''_{\mathcal{A}}(t_2) \leq V''_{\mathcal{A}}(t_1+\Delta t) - V''_{\mathcal{A}}(t_1) \qquad (19)$$

Since the time cost function is concave up, it is guaranteed that for any future time $t_2 > t_1$:

$$Cost([t_2,t_2+\Delta t]) \geq Cost([t_1,t_1+\Delta t]) \qquad (20)$$

Hence, it is guaranteed that for any future time t_2:

$$VOC = V''_{\mathcal{A}}(t_2+\Delta t) - V''_{\mathcal{A}}(t_2) - Cost([t_2,t_2+\Delta t]) \leq 0 \qquad (21)$$

And therefore termination at the current time is an optimal decision. □

Summary

The monitoring problem has been examined in two types of domains. One type is characterized by the predictability of utility change over time. High predictability of utility allows an efficient use of contract algorithms modified by various strategies for contract adjustment. The second type of domain is characterized by rapid change and a high level of uncertainty. In such domains, monitoring must be based on the use of interruptible algorithms and the value of computation criterion. In domains with moderate change and some degree of predictability of future utility, one can use an integrated approach. That is, activate the system with an initial contract time and then, if additional time is available, continue to monitor it using the value of computation criterion.

VI. Applications

The advantages of compilation and monitoring of anytime algorithms have been demonstrated through a number of applications. In this section we briefly describe two such application.

Mobile robot navigation

One of the fundamental problems facing any autonomous mobile robot is the capability to plan its own motion using noisy sensory data. A simulated robot navigation system has been developed by composing two anytime modules [22]. The first module, a vision algorithm, creates a local domain description whose quality reflects the probability of correctly identifying each basic position as being free space or an obstacle. The second module, a hierarchical planning algorithm, creates a path between the current position and the goal position. The quality of a plan reflects the ratio between the shortest path and the path that the robot generates when guided by the plan.

Anytime hierarchical planning is based on performing coarse-to-fine search that allows the algorithm to find quickly a low quality plan and then repeatedly refine it by replanning a segment of the plan in more detail. Hierarchical planning is complemented by an execution architecture that allows for the execution of abstract plans – regardless of their arbitrary level of detail. This is made possible by using plans as advice that direct the base level execution mechanism but does not impel a particular behavior. In practice, uncertainty makes it impossible to use plans except as a guidance mechanism.

The conditional performance profile of the hierarchical planner is shown in Figure 6. Each curve shows the expected plan quality as a function of run-time for a particular quality of the vision module. Finally, an active monitoring scheme was developed to use the compiled performance profile of this system and the time-dependent utility function of the robot in order to allocate time to vision and planning so as to maximize overall utility.

One interesting observation of this experiment was that the anytime abstract planning algorithm produced high quality results (approx. 10% longer than the optimal path) with time allocation that was much shorter (approx. 30%) than the total run-time of a standard search algorithm. This

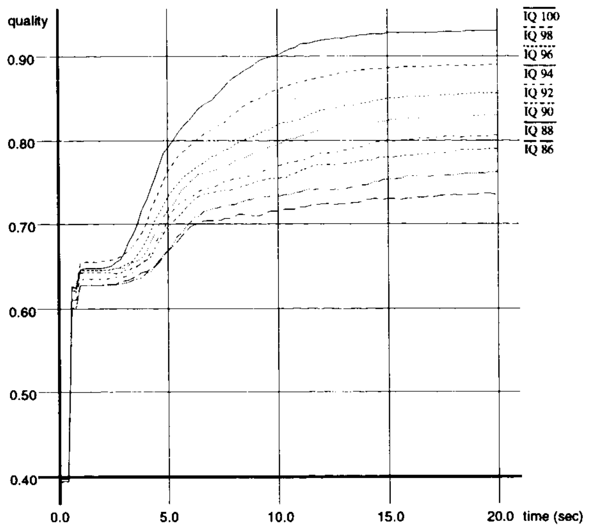

Figure 6: The CPP of the anytime planner

shows that the flexibility of anytime algorithms does not necessarily require a compromise in overall performance.

Model-based diagnosis

Model-based diagnostic methods identify defective components in a system by a series of tests and probes. Advice on informative probes and tests is given using diagnostic hypotheses that are based on observations and a model of the system. The goal of model-based diagnosis is to locate the defective components using a small number of probes and tests.

The General Diagnostic Engine [5] (GDE) is a basic method for model-based diagnostic reasoning. In GDE, observations and a model of a system are used in order to derive *conflicts* (A conflict is a set of components of which at least one has to be defective). These conflicts are transformed to *diagnoses* (A diagnosis is a set of defective components that might explain the deviating behavior of the system). The process of observing, conflict generation, transformation to diagnoses, and probe advice is repeated until the defective components are identified. GDE has a high computational complexity – $O(2^n)$, where n is the number of components. As a result, its applicability is limited to small-scale applications. To overcome this difficulty, Bakker and Bourseau have developed a model-based diagnostic method, called Pragmatic Diagnostic Engine (PDE), whose computational complexity is $O(n^2)$. PDE is similar to GDE, except for omitting the stage of generating all diagnoses before determining the best measurement-point. Probe advice is given on the basis of the most relevant conflicts, called *obvious* and *semi-obvious* conflicts (An obvious (semi-obvious) conflict is a conflict that is computed using no more than one (two) observed outputs).

In order to construct a real-time diagnostic system, Pos [13] has applied the model of compilation of anytime algorithms to the PDE architecture. PDE can be analyzed as a composition of two anytime modules. In the first module, a subset of all conflicts is determined. Pos implements this module by a contract form of breadth-first search. The second module consists of a repeated loop that determines which measurement should be taken next, takes that measurement and assimilates the new information into the current set of conflicts. Finally, the resulting diagnoses are reported.

Two versions of the diagnostic system have been implemented: one by constructing a contract algorithm and the other by making the contract system interruptible using our reduction technique. The actual slow down factor of the interruptible system was approximately 2, much better than the worst case theoretical ratio of 4.

VII. Conclusion

We presented a model for intelligent robot control that is based on compilation and monitoring of anytime algorithms. It offers both a methodological and a practical contribution to the field of real-time deliberation. The main aspects of this contribution include: (1) simplifying the design and implementation of complex intelligent robots by separating the design of the performance components from the optimization of performance; (2) mechanizing the composition process and the monitoring process; and (3) constructing machine independent real-time robotic systems that can automatically adjust resource allocation to yield optimal performance.

The study of anytime computation is a promising and growing field in artificial intelligence and in real-time systems. Some of the primary research directions in this field include: (1) Extending the scope of compilation by studying additional programming structures and producing a large library of anytime algorithms; (2) Extending the scope of anytime computation to include the two other aspects of robotic systems, namely sensing and action; and (3) Developing additional, larger applications that demonstrate the benefits of this approach. The ultimate goal of this research is to construct robust real-time systems in which perception, deliberation and action are governed by a collection of anytime algorithms.

References

[1] M. Boddy and T. L. Dean. Solving time-dependent planning problems. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pp. 979–984, Detroit, Michigan, 1989.

[2] M. Boddy. Anytime problem solving using dynamic programming. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pp. 738–743, Anaheim, California, 1991.

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

[4] T. L. Dean and M. P. Wellman. *Planning and Control*. San Mateo, California: Morgan Kaufmann, 1991.

[5] J. de Kleer and B. C. Williams. Diagnosing multiple faults. *Artificial Intelligence* **32**:97–130, 1987.

[6] A. Garvey and V. Lesser. Design-to-time real-time scheduling. To appear in *IEEE Transactions on Systems, Man and Cybernetics*, 1993.

[7] 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.

[8] E. J. Horvitz, H. J. Suermondt and G. F. Cooper. Bounded conditioning: Flexible inference for decision under scarce resources. In *Proceedings of the 1989 Workshop on Uncertainty in Artificial Intelligence*, pp. 182–193, Windsor, Ontario, 1989.

[9] E. J. Horvitz and J. S. Breese. *Ideal partition of resources for metareasoning*. Technical Report KSL-90-26, Stanford Knowledge Systems Laboratory, Stanford, California, 1990.

[10] R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence* **27**: 97–109, 1985.

[11] V. Lesser, J. Pavlin and E. Durfee. Approximate processing in real-time problem-solving. *AI Magazine* **9**(1):49–61, Spring 1988.

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

[13] A. Pos. *Time-Constrained Model-Based Diagnosis*. Master Thesis, Department of Computer Science, University of Twente, The Netherlands, 1993.

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

[15] S. J. Russell and S. Zilberstein. Composing real-time systems. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, pp. 212–217, Sydney, Australia, 1991.

[16] W-K. Shih, J. W. S. Liu and J-Y. Chung. Algorithms for scheduling imprecise computations with timing constraints. *SIAM Journal on Computing*, **20**(3):537–552, 1991.

[17] H. A. Simon. *Models of bounded rationality, Volume 2*. Cambridge, Massachusetts: MIT Press, 1982.

[18] J. von Neumann and O. Morgenstern. *Theory of Games and Economic Behavior.* Princeton, New Jersey: Princeton University Press, 1947.

[19] S. Zilberstein and S. J. Russell. Efficient resource-bounded reasoning in AT-RALPH. In *Proceedings of the First International Conference on AI Planning Systems*, pp. 260–266, College Park, Maryland, 1992.

[20] S. Zilberstein and S. J. Russell. Constructing utility-driven real-time systems using anytime algorithms. In *Proceedings of the IEEE Workshop on Imprecise and Approximate Computation*, pp. 6–10, Phoenix, Arizona, 1992.

[21] S. Zilberstein. *Operational Rationality through Compilation of Anytime Algorithms.* Ph.D. dissertation, Department of Computer Science, University of California at Berkeley, 1993.

[22] S. Zilberstein and S. J. Russell. Anytime sensing, planning and action: A practical model for robot control. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pp. 1402–1407, Chambery, France, 1993.