[Horvitz *et al.*, 1989] Eric J. Horvitz, H. Jacques Suermondt and Gregory F. Cooper. Bounded Conditioning: Flexible Inference for Decision Under Scarce Resources. In *Proceedings of 1989 Workshop on Uncertainty in Artificial Intelligence*, 1989.

[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.

[Lawler *et al.*, 1987] E. L. Lawler *et al.*, (Eds). *The traveling salesman problem: a guided tour of combinatorial optimization*. Wiley, New York, 1987.

[Russell and Wefald, 1991] Stuart J. Russell and Eric H. Wefald. *Do the Right Thing: Studies in limited rationality*. MIT Press, Cambridge, Massachusetts, 1991.

[Russell, 1991] Stuart J. Russell. An architecture for bounded rationality. In *Proceedings of the AAAI Spring Symposium on Integrated Architectures for Intelligent Agents*, Stanford, California, 1991.

[Russell and Zilberstein, 1991] Stuart J. Russell and Shlomo Zilberstein. Composing Real-Time Systems. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, Sydney, Australia, 1991.

[Zilberstein, 1991] Shlomo Zilberstein. Integrating Hybrid Reasoners through Compilation of Anytime Algorithms. In *Proceedings of the AAAI Fall Symposium on Principles of Hybrid Reasoning*, pages 143-147, Pacific Grove, California, November 1991.

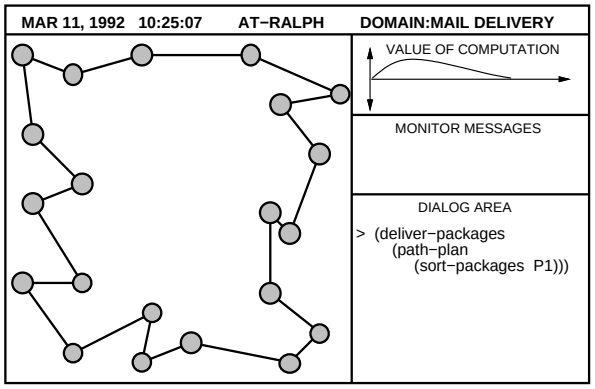


Figure 8: The user interface

will be able to load a particular domain definition and then run the system by specifying a particular problem using the dialog area. The system will then simulate the physical domain and the agent's behavior. The value of computation and a visual representation of the physical domain will be displayed during the simulation.

5 Conclusion

We have presented the three layers of AT-RALPH that together offer an integrated programming environment for constructing utility-driven, real-time agents. The central reasoning component of AT-RALPH is composed of elementary anytime algorithms that are combined together by the compiler into one anytime decision procedure. Our efficient meta-level control solves the problem of scheduling the elementary anytime components so as to maximize the overall performance. Laffey *et al* [1988] claim that "ad hoc techniques are used for making a system produce a response within a specified time interval". Our approach has many 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 effectively integrates planning and scheduling; and it provides machine independent real-time modules.

5.1 Further work

There is still a significant amount of system work to be done to generalize the various components of AT-RALPH. Another important task is to expand the "anytime" concept to include sensing and action. Moving toward a target in order to get a better view and aiming a gun at a target are examples of interruptible anytime actions. Similarly an anytime sensing procedure to estimate the size and location of an object can be developed by varying the number of samples and their resolution.

There is however an important distinction between sensing and action. Sensing may be assumed to have no relevant effect on the state of the domain while actions are intended to transform the state of the domain. As a result, it is possible to treat an anytime sensing component just as anytime computation and to apply to it the same compilation method. The contract-to-interruptible conversion cannot apply to actions, however, because re-initiating the action may result in different effects because of the changed initial state (consider, for example, gem polishing with increasingly fine abrasives). Further work is needed in order to integrate sensing and action into the framework. Our ultimate goal in this project is to construct an integrated programming environment for developing real-time agents that act by performing anytime actions and make decisions using anytime computation.

Acknowledgements

Support for this work was provided in part by the National Science Foundation under grants IRI-8903146 and IRI-9058427 (Presidential Young Investigator Award). The first author was also supported by the Malcolm R. Stacey Scholarship.

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] Thomas Dean and Mark Boddy. An Analysis of Time-Dependent Planning. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 49–54, Minneapolis, Minnesota, 1988.

[Doyle, 1990] Jon Doyle. Rationality and its Roles in Reasoning. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 1093–1100, Boston, Massachusetts, 1990.

[Elkan, 1990] Charles Elkan. Incremental, Approximate Planning: Abductive Default Reasoning. In *Proceedings of the AAAI Spring Symposium on Planning in Uncertain Environments*, Stanford, California, 1990.

[Hendler, 1989] James A. Hendler. Real Time Planning. In *Proceedings of the AAAI Spring Symposium on Planning and Search*, Stanford, California, 1989.

[Horvitz, 1987] Eric 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.

use them to generate the performance profile of the complete system and the monitor can use them for a quick evaluation of the utility of continued computation. This utility, also defined as the *value of computation*, is determined in the following way:

Given an algorithm $\mathcal{A}$, let S_{t_i} represent the state of the domain at time t_i, let q_{t_i} represent the quality of the results of the top-level interruptible anytime algorithm at time t_i. $U_{\mathcal{A}}(S, q)$ represents the utility of results of quality q in state S. The purpose of the meta-level control is to maximize $U_{\mathcal{A}}(S_{t_i}, q_{t_i})$. Due to uncertainty concerning the quality of the results of the algorithm, the expected utility of the results in a given state S at some future time t_i is computed by:

$$U'_{\mathcal{A}}(S, t_i) = \sum_q p(q_{t_i} = q) U_{\mathcal{A}}(S, q)$$

The probability $p(q_{t_i} = q)$ is provided by the performance profile of the algorithm. Due to uncertainty concerning the future state of the domain, the expected utility of the results at some future time t_i is computed by:

$$U''_{\mathcal{A}}(t_i) = \sum_S p(S_{t_i} = S) U'_{\mathcal{A}}(S, t_i)$$

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

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

In some cases it is possible to separate the utility of the results from the time used to generate them. In such cases the value of computation can be expressed simply by:

$$VOC = U'''_{\mathcal{A}}(q_{t+\Delta t}) - U'''_{\mathcal{A}}(q_t) - TC(t, \Delta t)$$

where $U'''_{\mathcal{A}}(q)$ is the utility of results of quality q, and $TC(t, \Delta t)$ is the time cost of Δt time units at time t. This assumption, when valid, simplifies the calculation of the value of computation.

4 The run-time system

The purpose of the run-time system is primarily to monitor the execution of the compiled anytime algorithm. It also provides a standard interface between the implemented agent, the physical environment, and the user interface. Figure 7 shows the data flow between the main components of the run-time system. Sensory input is used to update the state of the environment as perceived by the agent. The state is then used in order to compute the cost of time that, together with the current best results and the performance profile of the main decision procedure, is used to determine the value of computation.

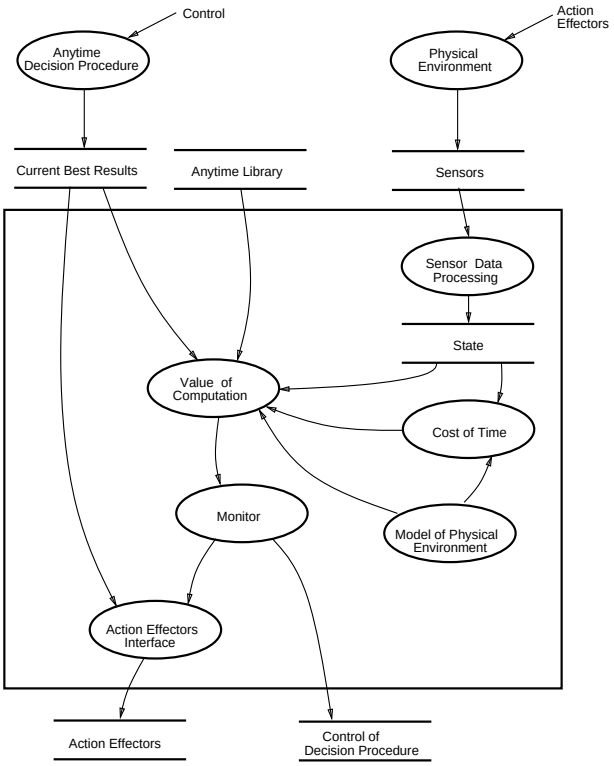


Figure 7: Data flow diagram of the run-time system

The monitor decides, based on the value of computation, whether to interrupt the decision procedure and apply the current best results. The system has two different modes: (1) Single decision mode; and (2) Continuous decision mode. In the first case the anytime algorithm is provided with a single set of inputs. It computes the results and may be interrupted by the monitor when the value of computation becomes negative. In the second case the algorithm is provided with an *input generator* that generates many instances of a problem. The run-time system restarts the algorithm with a new instance of the problem after each interruption/termination. This case models the behavior of an autonomous robot optimizing performance over a "history".

Another component of the run-time system, to be added in the future, is a debugging tool to allow the programmer to perform special operations that are unique to anytime computation such as: interrupting an algorithm and forcing its termination, examining the best result so far while the algorithm is running, or initiating events that change the cost of time to see how they affect the behavior of the monitor and the agent.

Figure 8 shows how the user interface of AT-RALPH will look in its complete implementation. The user

DAG representing of **F**. The task of the compiler in this case is to find for each total allocation of time t, and input quality Q_{in}, the run-times t_1, t_2, t_3, t_4 ($t_1+t_2+t_3+t_4 = t$) for the modules **G**, **H**, **I**, and **J** that maximize the expected quality of results Q_{out}. For each given allocation of time it is simple to find the expected quality of results based on the DAG representation and the performance profiles of the elementary anytime functions. In order to find the optimal allocation we have implemented an efficient search algorithm described below.

The time allocation algorithm is based on hill-climbing search. It starts with an equal amount of time allocated to each anytime component. Then it considers trading s time units between two modules so as to increase the expected quality of the results. As long as it can improve the expected quality, it trades s time units between the two modules that have maximal effect on output quality. When no such improvement is possible with the current value of s, it divides s by 2 until s reaches a certain minimal value. At that point, it reaches a local maxima and returns the best time allocation it found. As with any hill-climbing algorithm, it suffers from the problem of converging on a local maxima.

Time Allocation Algorithm

for each $Q_{in} \in$ *QUALS-TABLE*
  for each $T \in$ *TIME-TABLE*
    $s \leftarrow$ *initial-resolution(T)*
    $t_i \leftarrow T/n \quad \forall i : 1 \leq i \leq n$
    repeat
      while $\exists i, j$ such that
        $E(Q_{out}(Q_{in}, t_1, ..., t_i - s, ..., t_j + s, ..., t_n))$
          $> E(Q_{out}(Q_{in}, t_1, ..., t_n))$
        let i, j be the ones that maximize $E(Q_{out})$
        $t_i \leftarrow t_i - s$
        $t_j \leftarrow t_j + s$
      end
      $s \leftarrow s/2$
    until $s <$ *min-time-resolution*
    CPP$[Q_{in}, T] \leftarrow [Pr(Q_{out}), t_1, ..., t_n]$
  end
end

Complexity

The current version of the system uses discrete performance profiles stored in a two dimensional table (with input quality and run-time as indices). Linear approximation is used to compute the performance distribution for points that do not match exactly one of the table entries. The allocation algorithm therefore has to fill-in this two dimensional table. For each entry in the table, the complexity of the algorithm is $O(Kn^2 logT)$, where K is a small constant representing the number of times s units of time are traded before s is divided by 2, n is the number of modules and T is the total run-time. Since the number of modules used to define a new function is normally small the overall complexity of the algorithm is dominated by the accuracy of the compiled performance profile (a system parameter) and is independent of the program itself.

Finally, the compiler inserts in the original function the necessary code to control internal time allocation. For this purpose, the compiler replaces each call to an anytime algorithm by an activation of the following form:

```
(at '<anytime-function> :time <run-time>)
```

It activates the algorithm `<anytime-function>` as a contract algorithm with time allocation `<run-time>`. The amount of time allocated to each module is determined at run time by the total allocation and a simple lookup in the compiled performance profile. For example, the function defined in Figure 6 would be redefined as:

```
(defun F (u &key iq time &aux x y z)
    (setf x (at '(G u) :time
      (TA CPP-F iq time 1)))
    (setf y (at '(H x) :time
      (TA CPP-F iq time 2)))
    (setf z (at '(I x) :time
      (TA CPP-F iq time 3)))
    (at '(J y z) :time
      (TA CPP-F iq time 4)))
```

Note that `TA` is a general time allocation function that returns the appropriate time allocation based on the conditional performance profile of **F** (`CPP-F`), the initial input quality and total run-time.

3 Efficient meta-level control

Meta-level control of reasoning has been a leading technique in implementations of bounded rationality [Dean and Boddy, 1988; Doyle, 1990; Horvitz, 1987; Russell and Wefald, 1991]. In an early stage of these implementations it became apparent that gathering the necessary meta-level knowledge is a complicated task that can dominate the base-level problem itself. It was necessary therefore to develop a system in which the meta-level knowledge could be gathered automatically. Another goal was to be able to solve the meta-level decision problem quickly. This is exactly what we achieve by using conditional performance profiles. The compiler can automatically

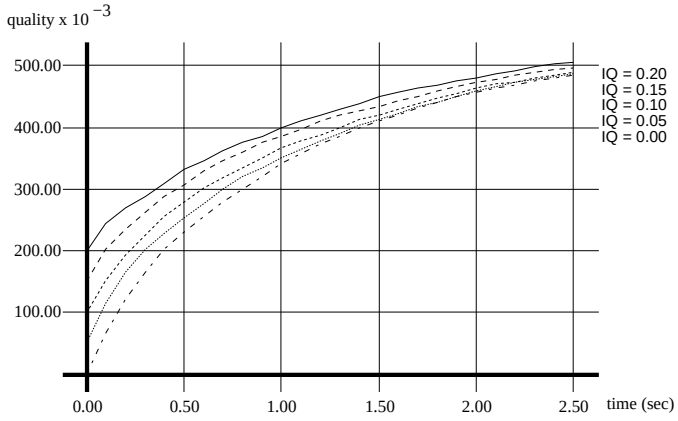


Figure 5: Conditional performance profile of TSP

achieved with run-time t. These statistics form the basis for the construction of the performance profile of the algorithm.

Quality measures

Russell and Zilberstein [1991] present three types of performance profiles and argue that the *performance interval profile* is the most appropriate for integrating anytime algorithms. However, for the purpose of this paper it will be easier for the reader to think about a performance profile as a mapping from computation time to the *expected* quality of results. This quality can be measured in three different ways: *Certainty* – where probability of correctness determines quality; *Accuracy* – where error bound determines quality; or *Specificity* – where the amount of detail determines quality. In principle, performance profiles can also be multidimensional, expressing, for example, both the certainty and accuracy of the results. However, we do not support multidimensional performance profiles in the current version of AT-RALPH.

Conditional performance profiles

To be able to properly combine anytime algorithms one has to take into account the fact that the quality of the results depends not only on time allocation but also on properties of the input, most notably its quality. Performance profiles in our system are therefore conditional. They consist of mappings from input quality and run-time to probability distribution of output quality:

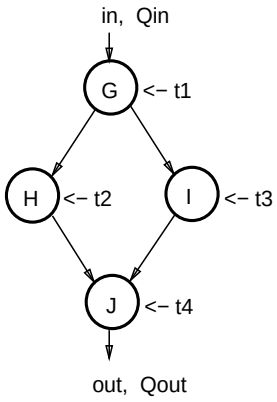
$$\mathcal{CPP} \ : \ Q_{in} \times T \rightarrow Pr(Q_{out})$$

Figure 5 shows the conditional performance profile that we got for the TSP algorithm with several different initial input qualities[4].

```
(defun F (u &aux x y z)
   (setf x (G u))
   (setf y (H x))
   (setf z (I x))
   (J y z))
```

```
(defun F (u)
   (J (H (G u)) (I (G u))))
```

6(a) Lisp definition of F



6(b) DAG representation of F

Figure 6: Compiling a straight line program

2.2 Compilation

In the current prototype of AT-RALPH we solved the compilation problem for the case where compound (non-elementary) anytime algorithms are restricted to composition of other anytime algorithms. An equivalent assumption is that the code of a compound algorithm can be written as a *straight line program*[5] with anytime algorithms as basic operations. For example, Figure 6(a) shows a definition of a compound anytime function **F** both as a straight line program (top version) and as a composition of anytime functions. Every straight line program has a corresponding DAG (Directed Acyclic Graph) representation where each node corresponds to one variable evaluation with a directed arc going to each function using that value. Figure 6(b) shows the

[4] Problems with a particular desired initial quality were generated using a different TSP algorithm.

[5] A straight line program is a sequence of expressions of the form *(setf u (f $v_1 ... v_n$))* where u is a new variable and $v_1 ... v_n$ are program arguments or existing variables. There is a trivial one-to-one mapping between functional composition and straight line code.

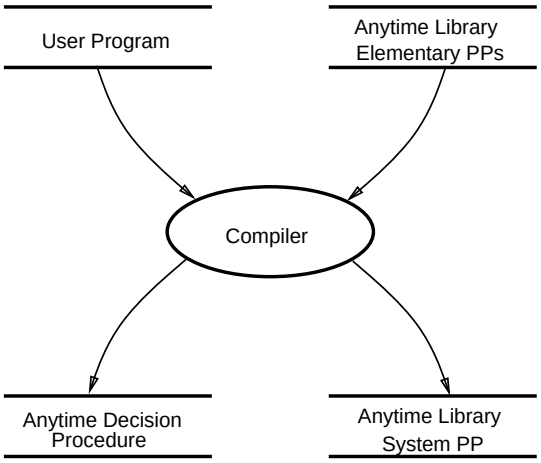


Figure 3: Compilation of decision procedures

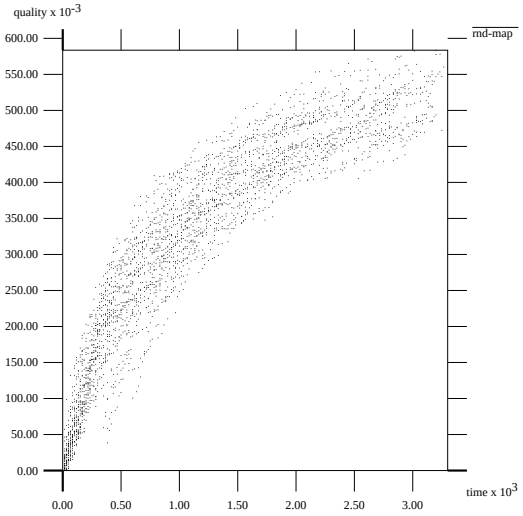


Figure 4: Quality map of TSP algorithm

struction. In addition, this constant can be reduced by scheduling the contract algorithm on a parallel machine. We have derived, for example, an optimal schedule that reduces the constant to 3 with 2 processors. The parallel scheduling options are, however, not trivial. We now turn to a detailed look at the compilation process.

Definition 1 Compilation of anytime algorithms *is the process of deriving a contract algorithm with an optimal performance profile from a program composed of several anytime algorithms whose performance profiles are given.*

Figure 3 shows the input and output of the compiler. Its input includes a user defined Lisp program composed of anytime components and a set of *conditional performance profiles* stored in a library. The task of the compiler is to produce a new version of the program that includes code to control the distribution of time between the components so as to maximize the overall performance for any given time allocation. It also creates a performance profile for the complete program based on optimal time allocation. The rest of this section describes the compilation process in detail.

2.1 Library of performance profiles

Once an elementary anytime algorithm is completed, its performance profile has to be computed and entered into the *anytime library*. The task of finding the performance profile of an algorithm can be quite complicated. In some cases, performance profiles can be constructed by a mathematical analysis of the anytime algorithm. For example in many iterative algorithms, such as Newton's method, the error in the result is (bounded by) a function of the number of iterations hence the performance profile can be constructed once the run-time of a single iteration is determined. In general, however, a mathematical analysis of the code is impossible. A general and practical approach is to compute the performance profile by gathering statistics on the performance of the algorithm in representative cases[2] (or learning it while the agent acts in the world). The performance profiles stored in the library can also be approximations to be refined within the context of a particular domain of application. At present we compute conditional performance profiles of the elementary anytime components by running a user-defined special program to gather the necessary statistics. In the future however, we plan to automate this task within our system.

We have implemented, for example, a randomized anytime algorithm for the Traveling Salesman Problem (TSP) that is based on tour improvement[3]. Quality in this case was measured by the percentage of tour length reduction (with respect to the original random tour). Figure 4 shows the quality map that we generated by running the algorithm with randomly generated input instances. Each point (t, q) represents an instance for which quality q was

[2] Representative problem instances are randomly generated based on prior knowledge of the problem domain.

[3] In the general case of tour improvement procedures, r edges in a feasible tour are exchanged for r edges not in that solution as long as the result remains a tour and the length of that tour is less than the length of the previous tour. See also [Lawler *et al.*, 1987].

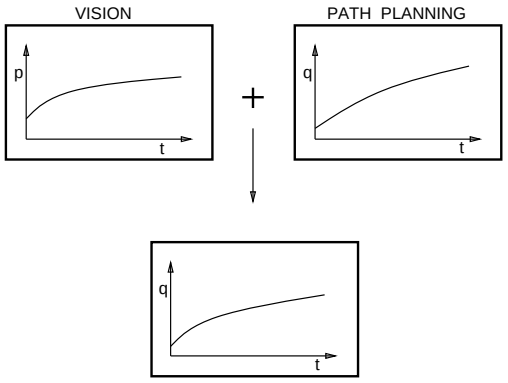


Figure 1: Path planning example

precision of the domain description.

Basic applications of anytime algorithms have been introduced by Boddy and Dean [1989] for solving a path planning problem and by Horvitz [1987] for real-time decision making in the health care domain. In this work we extend the use of anytime algorithms to the construction of complex planning systems. It is unlikely that a complex system would be developed by implementing one large anytime algorithm. Systems are normally built from components that are developed and tested separately. In standard algorithms, the expected quality of the output is fixed, so composition can be implemented by a simple call-return mechanism. However, when algorithms have resource allocation as a degree of freedom, incremental scheduling and constant monitoring are required to guarantee optimal utilization of resources.

Figure 1 illustrates our approach. It shows an anytime path planning algorithm that receives its input from an anytime vision module. The quality of vision is measured in terms of precision of the domain description. The quality of path planning is measured in terms of specificity of the suggested plan. Our compilation scheme combines these two modules into one anytime path planning algorithm that can automatically distribute any given amount of time between the two components so as to maximize the overall quality of results. The rest of this paper describes in detail our solution to the problem of integrating and controlling anytime algorithms. The result is an efficient and cheap meta-level control for real-time decision making that separates the performance components from the schedule optimization mechanism and automates the second task.

The next three sections of the paper correspond to the three conceptual layers of AT-RALPH as illustrated in Figure 2. In Section 2, we describe the compilation of anytime algorithms – a process that

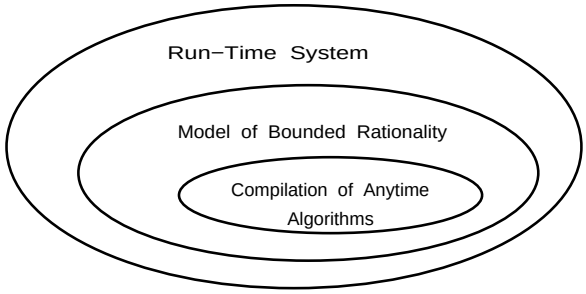


Figure 2: The three conceptual layers of AT-RALPH

creates an optimal anytime algorithm from a program composed of anytime components. Section 3 describes the theoretical framework used to determine the value of computation and the meta-level control of AT-RALPH. In Section 4, we describe the programming environment and run-time system. Finally, we summarize the benefits of our approach and discuss further work.

2 Compilation of anytime algorithms

The compilation of anytime algorithms, a central concept in our system, is an automated process that extends the idea of functional composition to anytime computation. Before explaining it in detail, we must make a distinction between *interruptible* algorithms and *contract* algorithms. Interruptible algorithms produce results of the quality "advertised" by their performance profiles 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. The greater freedom of design makes it easier to construct contract algorithms than interruptible ones. In fact the compilation process described below produces a contract algorithm. To make this algorithm interruptible we use the result from [Russell and Zilberstein, 1991]:

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

Note that $Q_{\mathcal{A}}(t)$ represents the actual quality of the results of $\mathcal{A}$ with time allocation t. The construction of the interruptible algorithm is based on repeatedly restarting the contract algorithm with exponentially increasing allocation of time. Some readers may wonder whether the constant, 4, could result in a significant degradation of performance. Our experience, however, shows that in dynamic situations the flexibility of using an anytime interruptible algorithm can offset the slowdown caused by its con-

To appear in Proceedings of the First International Conference on AI PLANNING SYSTEMS
June 15-19, 1992, College Park, Maryland

Efficient Resource-Bounded Reasoning in AT-RALPH

Shlomo Zilberstein and Stuart J. Russell
Computer Science Division
University of California
Berkeley, CA 94720

Abstract

Anytime algorithms have attracted growing attention in recent years as a key mechanism for implementing models of bounded rationality. The main problem, however, as with planning systems in general, is the integration of the modules and their interface with the other components of the system. We have implemented a prototype of AT-RALPH (Anytime Rational Agent with Limited Performance Hardware) in which an off-line compilation process together with a run-time monitoring component guarantee the optimal allocation of time to the anytime algorithms. The crucial meta-level knowledge is kept in the *anytime library* in the form of *conditional performance profiles*. These are extensions of an earlier notion of performance description – they characterize the performance of each elementary anytime algorithm as a function of run-time and input quality. This information, used by the compiler to produce the performance profile of the complete system, is also used by the run-time system to measure the *value of computation* and monitor the execution of the top-level procedure in the context of a particular domain. The result is an efficient and cheap meta-level control for real-time decision making that separates the performance components from the schedule optimization mechanism and automates the second task.

1 Introduction

Our objective has been to develop an efficient meta-level control for utility-driven, real-time agents. These agents are designed to operate in complex domains where future states and utility change over time are unpredictable. The complexity of these domains on the one hand and the limited computational resources available to the agents on the other hand have led us to a solution based on bounded rationality and anytime computation. Anytime algorithms[1] have attracted growing attention in recent years as a key mechanism for implementing models of bounded rationality. The main problem, however, as with planning systems in general, is the integration of the anytime components into a single, efficient, decision making procedure and interfacing this procedure with the other components of the agent – perception and action. In AT-RALPH, this problem is solved through an off-line compilation process together with a run-time monitoring component that guarantee the optimal allocation of time to the elementary anytime algorithms.

Anytime computation introduces a tradeoff between run-time and quality of results. In order to optimally control this new degree of freedom we introduce *conditional performance profiles* that give a probabilistic description of the quality of the results of an algorithm as a function of run-time and input quality (or a set of properties of the input). This is an extension of an earlier notion of performance profile that depends on run-time only.

Consider for example an anytime hierarchical planner whose quality of results is measured by the level of specificity of the plan. Obviously, the specificity of a plan affects its execution time and hence has influence on the efficiency of the agent. We have recently implemented such an algorithm for hierarchical path planning. Its conditional performance profile describes how the quality of the plan depends on run-time as well as on input quality, that is, the

[1] Anytime algorithms are algorithms whose quality of results improves gradually as computation time increases.