Controlling generate & test in any time

Coulon, C.; van Harmelen, F.A.H.; Karbach, W.; Voss, A.

Controlling generate & test in any time

Carl-Helmut Coulon[◇] and Frank van Harmelen[°] and Werner Karbach[◇] and Angi Voß[◇]

[◇]German National Research Institute for Computer-Science (GMD)
P.O. Box 1316, D-5205 Sankt Augustin
[°]University of Amsterdam
Roeterstraat 15, NL-1018 WB Amsterdam
e-mail: avoss@gmdzi.gmd.de

Abstract. Most problem solvers have a one-dimensional stop criterion: compute the correct and complete solution. Incremental algorithms can be interrupted at any time, returning a result that is more accurate the more time has been available. They allow the introduction of time as a new dimension into stop criteria. We can now define a system's utility in terms of the quality of its results and the time required to produce them. However, optimising utility introduces a new degree of complexity into our systems. To cope with it, we would like to separate the performance system to be optimised from utility management.
Russell has proposed a completely generic precompilation approach which we show to be unsatisfactory for a generate & test problem solver. Analysing this type of systems we present four different strategies, which require different information and result in different behaviours. The strategy most suitable to our application requires on-line information, and hence had to be implemented by a meta-system rather than a precompiler. We conclude that universal utility managers are limited in power and are often inferior to more specialised though still generic ones[1].

1 Motivation

KBS must take time into account: Knowledge-based systems are often built to cope with really hard problems. Although AI is famous for tackling NP-hard problems, its systems usually do not consider the time that any benevolent user may be ready to wait. They are designed and fine-tuned to achieve a fixed stop criterion. For example, most diagnosis systems stop when they reach a leaf in the diagnosis hierarchy. However, if the problem space of such systems is equipped with a notion of incomplete, approximate or partial solutions, the user might sometimes prefer a quick, though approximate, solution.

[1] The research reported here was carried out in the course of the REFLECT project. This project is partially funded by the Esprit Basic Research Programme of the Commission of the European Communities as project number 3178. The partners in this project are the University of Amsterdam (NL), the German National Research Institute for Computer-Science GMD (D), the Netherlands Energy Research Foundation ECN (NL), and BSR Consulting(D).

Entering a new dimension of utility: Real time systems have recently been defined to be systems whose utility gracefully decreases the shorter they are run [2] [1]. They are based on incremental or interruptible anytime-algorithms whose quality of the output increases over time. These are by no means rare creatures. In principle, every loop is a candidate to be turned into an incremental procedure. Incremental, interruptible, or anytime algorithms introduce a new degree of freedom when defining system utility. Instead of concentrating on the ultimate, most correct solution, we can try to obtain maximal quality within a given time span, or try to reach a minimum quality as quickly as possible, or combine both. Thus, the utility of a system becomes a function of time U(t).

Utility should be handled orthogonally: This new degree of freedom must be used carefully, so as not to introduce a new dimension of complexity into our systems. Ideally, a KBS should be designed as before, and utility optimisation should be handled by a separate component. Beside the utility function to be achieved, such a utility manager needs certain information about the performance of the underlying system. If all relevant information is available before run time, a precompiler would do perfectly.

A utility manager for generate & test: Generate & test is a frequently employed problem solving method in AI: in diagnosis, hypotheses are generated and tested; in search, successor states are generated and evaluated; in design, solutions are proposed & revised; in planning plans are generated and tested by execution. In this paper, we present the principles of generic utility management for the class of generate & test problem solvers. We elaborate four strategies of switching between generation and test in order to maximise the number of solutions given a maximal time limit. The goal of this paper is *not* to present sophisticated strategies for controlling generate & test. In fact, with the possible exception of the fourth strategy, the strategies we discuss are rather obvious. Instead, the goal of this paper is to present a method of comparing such strategies, and to introduce the parameters that are involved in such comparisons.

A comparison to Russell and colleagues will round off the paper. Although developed independently, their motivation on utility is very similar to our's. In [4] they propose a precompiler separating the algorithmic design of real-time systems from optimising their utility. The latter task is automated based on so-called performance profiles for the basic algorithms. Although we appreciate their intention, we doubt the practicality of their assumptions. While they derive the overall performance profile of a system from those of its basic algorithms, we would like to specify the overall performance profile without having to supply any profiles for the basic steps. Moreover, we have several potential overall performance profiles, but cannot determine the right one statically, so that precompilation is not suitable. In the meantime Russel's & Zilberstein's system maintains several performace profiles and introduces a monitoring component which switches between them at run-time[2]. But we still see the problem of determining which profile to apply in a specific situation.

[2] Personal communication.

2 A utility function for generate & test

We consider a class of object systems that employ a generate & test problem solving method to produce all possible solutions.

Definition of generate & test algorithms: The generate-phase of such algorithms generates candidates for a full solution which are subsequently tested on their correctness. The characteristics that candidates for *full* solutions must be generated excludes an algorithm as propose & refine from our analysis since it proposes a *partial* solution which is subsequently refined on the basis of the test results.

Importance of generate & test algorithms: Although generate & test algorithms are among some of the oldest AI, and feature in every text book, they are in general regarded as not very efficient. It is true that non-heuristic generate & test algorithms of the kind discussed in this paper do not scale up to very large search spaces. Nevertheless, generate & test algorithms were the basis of such programs as DENDRAL [3], one of the few successful early AI programs that were ever used in practice. The crucial insight there was to first use a planning process that uses constraint-satisfaction techniques to create lists of recommmended candidates. The generate & test procedure than uses those lists so that it can explore only a fairly limited set of candidates. This shows that even the fairly simple version of non-heuristic generate & test that we study in this paper can be of interest in full scale application systems.

Both generate and test are conceived as incremental algorithms that can be called repeatedly in order to generate resp. test a next hypothesis. We thus have the following data flow of the underlying system:

$$input \Longrightarrow GENERATE \Longrightarrow hypotheses \Longrightarrow TEST \Longrightarrow solutions$$

As a stop criterion we want to impose upon such systems a lower bound on the number of solutions to be produced, and an upper bound on the time to be spent:

$$stop(sol, t) := sol \geq sol_{min} \vee t \geq t_{max} \quad (1)$$

with	sol	:= number of solutions produced by the system
	sol_{min}	:= minimum number of solutions desired
	t	:= time needed by the system
	t_{max}	:= upper bound on the run-time of the system

The parameterised stop criterion specifies the minimally required quality of the system output. Often, only one solution may be necessary (i.e. $sol_{min} = 1$), but sometimes the system does not have all knowledge, e.g. because it is too difficult to represent. Then the human user may want to see alternative solutions to choose among using his additional knowledge, which may be too difficult to represent in the system. We have also considered other stop criteria: simpler ones such as $stop(sol) := sol \geq sol_{min}$, and more complex ones that additionally impose an upper limit on the number of solutions desired. Stop criterion (1 is a moderately complex one, whose analysis is sufficiently interesting for the purpose of this paper.

The stop criterion alone does not yet ensure the desired behavior. For instance, in formula (1) the system might just wait until the given time has passed (i.e. until $t > t_{max}$). The precise goal of the system is to reach the minimally required quality *as fast as possible* within the given time. That means, we want to minimise the time and maximise the number of solutions while observing the stop criterion:

$$U(sol,t) := \max\{\frac{sol}{t} | stop(sol,t)\} \tag{2}$$

To analyze this function we will refine its parameters, computation time t and solutions sol. In the most general case, the generate method may be invoked several times, followed by some invocations of the test method, and these two phases may be iterated a number of times. Additionally, switching between the two methods may cost additional time to store respectively reinstall the current state. The time spent is thus defined by:

$$t := \sum_{i=1}^{n}(\sum_{j=1}^{h_{gen_i}} t_{gen_{ij}} + t_{switch_i}^{gen\mapsto test} + \sum_{j=1}^{h_{test_i}} t_{test_{ij}} + t_{switch_i}^{test\mapsto gen}) \tag{3}$$

with

n	:= the number of times the system switches from generate to test;
h_{gen_i}	:= the number of hypotheses generated in the ith call to generate;
h_{test_i}	:= the number of hypotheses tested in the ith call to test;
$t_{gen_{ij}}$	:= time to generate one hypothesis;
$t_{test_{ij}}$	:= time to test one hypothesis;
$t_{switch_i}^{gen\mapsto test}$	:= time to switch from generate to test;
$t_{switch_i}^{test\mapsto gen}$	:= time to switch from test back to generate with $t_{switch_n}^{test\mapsto gen} = 0$.

The number of solutions produced by the system corresponds to the sum of the probabilities p_{ij} of all tested hypotheses:

$$sol := \sum_{i=1}^{n} \sum_{j=1}^{h_{test_i}} p_{ij} \tag{4}$$

3 Analysis of four strategies for utility management

Simplifying assumptions. Before we will discuss four different strategies for controlling a generate & test system, we will introduce some assumptions that will simplify equations (3) and (4). We will assume that all generated hypotheses are tested (A1), that the times for switching from generation to test and vice versa are constant and equal (A2), that the time to generate resp. test a hypothesis are equal for all hypotheses (A3), and that solutions are distributed uniformly among the hypotheses (A4).

$$\begin{aligned}
&(A1) \quad \forall i, 1 \leq i \leq n : h_{gen_i} = h_{test_i} = h_i \\
&(A2) \quad \forall i, 1 \leq i \leq n : t_{switch_i}^{gen\mapsto test} = t_{switch_i}^{test\mapsto gen} = \frac{t_{switch}}{2} \\
&(A3) \quad \forall i, j : t_{gen_{ij}} = t_{gen}; t_{test_{ij}} = t_{test} \\
&(A4) \quad \forall i, j : p_{ij} = p
\end{aligned}$$

Justifying the assumptions Assumption (A1) no longer allows us to test only the most promising hypotheses. This assumption is automatically fulfilled in domains where all hypotheses *must* be tested, for instance because an exhaustive solution is required, or because the best solution is required. In many domains, no easy ranking of the hypotheses is possible, or more precisely: such ranking is often considered to be part of the test phase of the system, rather than as a way to control the behaviour of the overall cycle.

Assumption (A2), forces switching in both directions to be equally expensive and constant for all cycles. It seems rather realistic to assume switching time to be constant, since the switching cost is likely to be independent from the particular hyptheses that have just been generated or tested. The assumption that switching times in both directions are equal could easily be dropped, and our model could be trivially extended to deal with different switching times in both directions. We will however not present this extension in this paper since it only complicates matters without offering any new insights.

Besides motivating the constant and equal values of the switching times, we should also motivate why we consider switching time at all, in other words, why would $t_{switch} > 0$? The value of t_{switch} should be interpreted as the overhead of starting a new series of generating or testing steps, and there are many applications in which this overhead is indeed a considerable factor. In medical or mechanical diagnosis forinstance, the generation of new hypotheses often involves new measurements (on a patient or a device), and the overhead of starting a new series of observations (getting the patient in the lab, or halting and opening the machine) is often high compared to the cost of making the observations themselves.

Assumption (A3) states that generation and test times are equal for all hypotheses. This is an assumption that will hold in some domains and not in others. In game playing for instance, the costs of generating new board positions and evaluating them are indeed roughly independend from the particular board position. In other domains however, the testing time in particular is likely to vary for different hypotheses: the cost of testing solutions to a design problem may vary significantly across different solutions, because inconsistencies with the design constraints may show up immediately or only very late during the testing phase.

In such case, the parameter t_{test} should be regarded as the "average" cost of testing a hypothesis.

Of all our assumptions, (A4) is the most restrictive. It assumes that solutions are uniformly distributed across the hypothesis space, and this will often not be the case in realistic applications. In game playing for instance, the entire section of the search space below a losing move will be devoid of solutions, making the value of p in that section of the search space much lower then in other sections. It is mainly because of this assumption that our model must be seen as a first approximation of the behaviour of real systems, rather than as a model that captures the precise behaviour of these systems.

Applying the assumptions A1-A3 simplify equation (3) for the time required by the system to make n iterations as follows[3]:

$$t := \sum_{i=1}^{n} (h_i \cdot (t_{gen} + t_{test})) + n \cdot t_{switch} \tag{5}$$

We will now define different strategies for the generate & test algorithm by giving different definitions for the numbers h_i in this equation. They will lead to a different switching behavior between the generate and test phases. We will first concentrate on the time required to met the fist condition of the stop criterion ($sol \geq sol_{min}$) and in section 5 compare the number of solutions if $t \leq t_{max}$ is reached first.

The first part of the stop criterion requires that we compute at least sol_{min} solutions. This implies that the expected number of iterations between generate and test that are to be made in order to achieve the stop criterion is the lowest number n such that

$$sol_{min} \leq p \cdot \sum_{i=1}^{n} h_i \tag{6}$$

3.1 Strategy 1 - directly generate the right number of hypotheses

If we assume we know the probability p of a generated hypothesis to pass the test, we can estimate how many hypotheses we will need to obtain sol_{min} solutions in the first iteration, namely $\frac{sol_{min}}{p}$ hypotheses. There will be no need to switch back:

(S1) $h_1 = \lceil \frac{sol_{min}}{p} \rceil$ implying $n = 1$.

[3] This formula is not entirely correct since it assumes a last switch back from test to generation. To obtain the correct times for the strategies, half of t_{switch} should be subtracted in the time formulae given below. But we preferred to keep our formulae more readable, and the constant does not affect our comparison of the strategies.

This means that the expected time needed to compute sol_{min} solutions will be:

$$t_{sol_{min}} = \lceil \frac{sol_{min}}{p} \rceil \cdot (t_{gen} + t_{test}) + t_{switch} \tag{7}$$

This strategy is optimal since both the number of hypotheses generated and the number of switches is minimal. To implement the strategy, only h_1 must be computed, which can be done statically. The major problem with this strategy is of course that the probability p of a generated solution to pass the test is often not known.

3.2 Strategy 2 - eager generation:

This strategy exhaustively generates all possible hypotheses in the first call to generate (h_1), and then tests all of them. It, too, does not switch back. If we write h_{all} for the number of all possible hypotheses and sol_{all} for the number of all possible solutions, this strategy is defined by:

(S2) $h_1 = h_{all} = \lceil \frac{sol_{all}}{p} \rceil$ implying $n = 1$.

The formula for the runtime of the system is:

$$t_{sol_{min}} = \lceil \frac{sol_{all}}{p} \rceil \cdot (t_{gen} + t_{test}) + t_{switch} \tag{8}$$

The number of switches is minimal, but usually too many hypotheses are generated, since $sol_{all} \geq sol_{min}$, causing S2 to be more expensive than S1. Therefore, S2 can be recommended only when switching costs are very high, and $sol_{all} \approx sol_{min}$, so that not too many unnecessary hypotheses are generated. The advantage is that we need not have to know p. Notice that this strategy assumes that h_{all} is finite (since otherwise the first phase of the algorithm never terminates).

3.3 Strategy 3 - lazy generation:

The third strategy generates hypotheses one by one and directly tests each:

(S3) $\forall i : h_i = 1$ implying $n = \lceil \frac{sol_{min}}{p} \rceil$

The expected time to compute sol_{min} hypotheses is:

$$t_{sol_{min}} = \lceil \frac{sol_{min}}{p} \rceil \cdot (t_{gen} + t_{test}) + \lceil \frac{sol_{min}}{p} \rceil \cdot t_{switch} \tag{9}$$

This strategy will not generate unnecessary hypotheses, but abounds in switches. It can be recommended only when switching time is low. If $t_{switching} = 0$ its behavior is equal to S1 and hence optimal.

3.4 Strategy 4 - generate the number of missing solutions:

The fourth strategy always generates as many hypotheses as there are solutions still missing:

$$(S4)\ \forall i : h_i = sol_{min} - sol_{i-1}$$

where sol_i is the total number of solutions found after completing the ith iteration and $sol_0 = 0$.

Applying (A1) and (A4) to (4) gives $sol_i = \lceil p \cdot \sum_{k=1}^{i} h_k) \rceil$ which leads to $(1-p)^{i-1} sol_{min}$ as an approximation for h_i, with which we can derive the following approximation for the expected run time:

$$t_{sol_{min}} \approx \frac{sol_{min}}{p} \cdot (1-(1-p)^n) \cdot (t_{gen} + t_{test}) + n \cdot t_{switch} \qquad (10)$$

Again no superfluous hypotheses are generated. S4 will always behave at least as good as S3 because in comparing (10) and (9) we see that $(1-(1-p)^n) < 1$, and $n \leq sol_{min} \leq \frac{sol_{min}}{p}$. To implement S4 we have to compute the numbers h_i dynamically, whereas these numbers could be computed statically for S1-S3.

4 Comparison of the strategies when reaching $sol \geq sol_{min}$ first

S1 has optimal run-time, but requires knowledge of p, which is usually not available. It requires no additional computation during the execution of the generate & test algorithm.

S2 guarantees that $n = 1$, and is therefore good for $t_{switch} \gg (t_{gen} + t_{test})$. The extra costs of S2 are limited if $sol_{all} \approx sol_{min}$. S2 does not require p, and involves no additional computations.

S3 is the opposite of S2. S3 switches many times, and is therefore only good for $t_{switch} \ll (t_{gen} + t_{test})$ (S3 is in fact optimal if $t_{switch} = 0$). Again, as with S2, S3 requires neither p nor any additional computation.

S4 is a compromise strategy: it makes more switches than S2 but less than S3, it generates more hypotheses than S3 but less than S2, and its run-time is more than S1 but less than S3. However, S4 is the only strategy which requires an additional, though simple computation of h_i as $sol_{min} - sol_{i-1}$.

Thus, the choice of strategy depends on p and on the ratios $t_{switch} : (t_{gen} + t_{test})$ and $sol_{all} : sol_{min}$. We have an optimal strategy only if p is known.

5 Comparison of the strategies when reaching $t \leq t_{max}$ first

So far, we have compared the strategies S1-S4 on the basis of their overall run-time. There is, however, another dimension along which we can compare them, namely on the basis of how uniformly they compute their solutions over time. This is important because the stop criterion (1) says that the system will stop when t_{max} has been reached, which may be before sol_{min} solutions have been computed (if $t_{max} < t_{sol_{min}}$). In general, we will not know this inequality in advance, since $t_{sol_{min}}$ depends on p which may not be known. Because of this, it becomes important that the composite behavior produced by the strategy is *interruptible* [4]. Below we will investigate this propery for each of the strategies S1-S4. We will do this on the basis of the graphs in figure 1, which indicate for each strategy how the computation of solutions proceeds over time. For each of these strategies, we will establish the number of solutions produced per time-unit, in other words: $sol_i/time$.

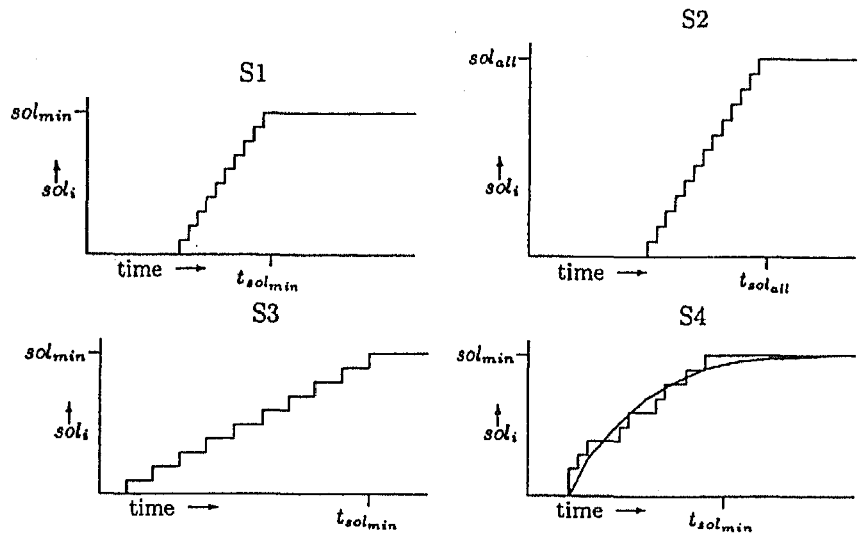

Fig. 1. Distribution of solutions over time for S1-4

S1, S2 will not produce any solutions for a long time, and then suddenly produce all solutions at once, namely after the first (and final) call to *test*. This means that $sol_i/time = 0$ for a long time, and then jumps to sol_{min} at $t_{sol_{min}}$ for S1 and to sol_{all} at $t_{sol_{all}}$ for S2.

S3 produces solutions incrementally, and at a constant rate, namely on the average 1 solution per $1/p$ iterations. Thus, the value of $sol_i/time$ is constant over time, at $p/(t_{gen} + t_{test} + t_{switch})$.

S4 also produces its solutions incrementally, but in ever decreasing chunks: at each iteration, nothing happens for $h_i \cdot (t_{gen} + t_{test}) + t_{switch}$ time units, and then $h_i \cdot p$ solutions are produced. Thus, the number of solutions per time is a step function with a decreasing angle, namely $p \cdot (t_{gen} + t_{test} + (t_{switch}/h_i))^{-1}$, and with ever decreasing steps both horizontally, namely $h_i \cdot (t_{gen} + t_{test}) + t_{switch}$, and vertically, namely $h_i \cdot p$). This step function can be approximated by a function asymptotically approaching sol_{min}.

We are now in a position to compare the different strategies with respect to their production of solutions over time.

S1 is a strategy with minimal run-time, but is not interruptible: if it gets interrupted because $t_{max} < t_{sol_{min}}$, we get no solutions at all. Thus, S1 is a *high-risk/high-pay* strategy.

S2 is not interruptible either. It takes longer than S1, but ensures that no switching back is required. We get no solutions when it is interrupted.

S3 on the other hand is an anytime algorithm, which will have produced some solutions when it gets interrupted prematurely. However, this is at the price of making many switches. Thus, S3 is *low-risk/low-pay*.

S4 is a compromise strategy. In the beginning, it looks like S1 and later on more like S3. This gradual change in behavior from a high-pay/high risk strategy to a low-pay/low-risk one makes sense, since the chance of running into t_{max} increases with time.

6 Problems with Russell's approach

In [4] the authors presented an approach for utility management by composing elementary anytime algorithms. To cite from their paper: "...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"[4]. The time allocated to the primitive routines are pre-compiled from the utility function $U^*(t)$ of the composite system which is determined by the individual PPs $U_i(t)$ of the basic components.

How can we accomplish our utility function and strategies in this framework? First of all, we have to compose the generate & test method from the generate, switch and test steps: `(LOOP generate switch test switch)`.

Next we have to define utility in terms of the PPs for the basic steps. However, it turns out that we cannot come up with a unique PP for the generate and the test steps because they are highly dependent on the input, in particular on the probability p of hypotheses being solutions, which cannot be estimated statically in every application. Figure 2 shows the bands of potential PPs of the basic inference steps. This is why we doubt Russel and Zilberstein's basic assumption

[4] PP is short for performance profile.

that the input can be partitioned into classes whose elements have the same PP (p. 213 in [4]). However, if we know the probability p, their approach would indeed lead to the optimal strategy S1.

To cope with the missing information in our application, we chose the fourth strategy which allows us to replace p by information gathered on-line. We implemented it by a meta-system, since a precompiler as suggested by Russell and Zilberstein would not have done.

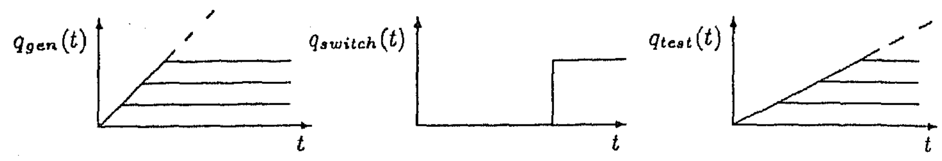

Fig. 2. Performance profiles for generate, switch & test

The main difference, however, between Russel's & Zilberstein's approach and ours is that they will always produce a single time schedule, while in our approach, choosing another strategy will result in different performance profiles of the compound system as depicted in figure 1. Moreover, to use their terminology, our strategies S1 and S2 result in *contract algorithms*, which must know the available time in advance, while S3 and S4 yield *interruptible algorithms* which produce meaningful results whenever they are interrupted and without being told when they will get interrupted. Their approach results in contract algorithms which then have to be transformed to interruptible ones slowing them down by factor 4 at most[5]. Our strategies S3 and S4 do not suffer from a slowdown.

Since our strategies depend on the information available, and differ in the overall performance profiles, in the risk vs. pay ratio, and in the interruptability of the composite system, our approach is much more flexible. This however, is at the cost of being specialised to generate & test methods.

7 Conclusion

For the class of generate & test methods we defined a utility function that involves a lower bound on the number of solutions and an upper bound on the time to be spent. We analyzed four different strategies and compared them with respect to the information needed, their temporal behaviors and their interrruptability. Our results should be easy to carry over to simpler or more complex utility functions.

[5] In the meantime, they have developed compilation methods for a loop which directly lead to interruptible algorithms.

References

1. T. Dean and M. Boddy. An analysis of time-dependent planning problems. In *Proceedings of the 7th National Conference on Artificial Intelligence*, volume 1, pages 49 - 54, San Mateo, 1988. Morgan Kaufmann.
2. E.J. Horvitz. Reasoning about beliefs and actions under computational resource constraints. In L.N. Kanal, T.S. Levitt, and J.F. Lemmer, editors, *Uncertainty in Artificial Intelligence 3*, pages 301–324. Elsevier, Amsterdam, 1987.
3. R.K. Lindsay, B.G. Buchanan, E.A. Feigenbaum, and J. Lederberg, editors. *Applications of Artificial Intelligence for Organic Chemistry: The DENDRAL Project*. McGraw-Hill, New York, 1980.
4. S.J. Russell and S. Zilberstein. Composing real-time systems. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence, Sydney, Australia*, volume 1, pages 212 - 217, San Mateo, 1991. Morgan Kaufmann.