Describing Problem Solving Methods using Anytime Performance profiles

Annette ten Teije* and Frank van Harmelen
Department of AI
Faculty of Science
Vrije Universiteit Amsterdam
{annette,frankh}@cs.vu.nl

Abstract

Traditional selection criteria for Problem Solving Methods (PSMs) from libraries concentrate solely on the functionality of these methods, and ignore their computational performance. We propose the use of anytime performance profiles to describe the computational behaviour of problem solving methods, and to use these as additional selection criteria when selecting methods from a library. A performance profile describes how the quality of the output of an algorithm gradually increases as a function of the computation time. Such anytime descriptions of problem solving methods are attractive because they allow a trade-off to be made between available computation time and output-quality. It turns out that many problem solving methods found in the literature have a natural anytime behaviour, which has remained largely unexploited until now.

In this paper we propose an axiomatic description of performance profiles. Furthermore, in order to make our proposal feasible for library builders, we give guidelines on how to organise such axiomatic descriptions. Finally, we apply our proposal to a number of realistic problem-solving methods, namely hierarchical classification (used in MDX), parametric design (methods from XCON and VT), and consistency-based diagnosis (the GDE-method).

* Supported by the Netherlands Computer Science Research Foundation with financial support from the Netherlands Organisation for Scientific Research (NWO), project: 612-32-006

The copyright of this paper belongs to the papers authors. Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage.

Proceedings of the IJCAI-99 workshop on Ontologies and Problem-Solving Methods (KRR5) Stockholm, Sweden, August 2, 1999

(V.R. Benjamins, B. Chandrasekaran, A. Gomez-Perez, N. Guarino, M. Uschold, eds.)

http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-18/

1 Motivation

One of the major themes in the literature on problem-solving methods (PSMs) of the past half decade has been the so-called *applicability problem*: how to decide which PSMs are applicable to a given task. Solving this problem is essential for delivering some of the promises made about PSM, in particular library construction and re-usability. Solving the applicability problem boils down to identifying a set of properties of PSMs in such a way that these properties can be used to select the appropriate methods from a library.

The literature contains many proposals on how to describe properties of PSMs, and we will not try to give a systematic overview of them. Many of the proposals are mentioned in [BPG96]. That paper synthesises all the different proposals and defines three categories of applicability conditions[1]: teleological conditions (= does the goal of the PSM match with the current task at hand) epistemological conditions[2] (= knowledge requirements of the PSM) and pragmatic conditions (= requirements on the interaction of the PSM with its environment).

All of the proposals on applicability conditions in the literature regard a PSM as a functional I/O relation between domain knowledge and goal, and formulate the method selection criteria in terms of this I/O relation: (a) does the goal of the PSM match the current task at hand, and (b) does the available domain knowledge match the knowledge requirements of the PSM. Much of the more recent

[1]We prefer the term *applicability conditions* over the term *assumptions*, since the properties concern conditions that must be tested, and not assumptions which can be assumed.

[2]later named ontological conditions in [FB98]

work on characterising PSMs still focuses exclusively on the so-called "competence"[3] of the PSM [WAS98]. This also holds for [FS98], which emphasises more than most the importance of computational behaviour besides only the functional I/O relation. Even that paper does not give a concrete proposal for how to describe the computational behaviour. The same holds for [FB98], which describes in some detail the effect that various conditions have on the computational behaviour of some diagnostic methods, but this analysis is all done informally, with no proposal on how to describe the computational behaviour of a method.

This leaves an entire dimension of PSM applicability conditions uncovered in the literature: *how should the performance of a PSM be described so that it can be used as an applicability condition?* There are good reasons why performance description of PSM has been left as an open question in the Knowledge Engineering literature: as argued in [FS98], the standard worst case complexity measures from computer science are not very helpful for PSMs, since a significant part of PSMs are of a heuristic nature, and the worst case complexity describes only the case when the heuristics do not apply. One of the few attempts at describing average case complexity is [SB95], but this approach requires detailed knowledge on the expected heuristic behaviour of all of the subtasks of a PSM, which does not seem very realistic in practice.

Instead, in this paper we propose the use of so-called *anytime performance profiles* [DB88] to describe the computational behaviour of PSMs (see section 2). Traditionally, such performance profiles are given in the form of graphs which are obtained empirically by executing the PSM. We propose an axiomatic description of performance profiles (section 4). Furthermore, in order to make our proposal feasible for library builders, we give guidelines on how to organise such axiomatic descriptions (section 4): our descriptions always consist of the same four elements, each of which must be filled in by the library-builder when characterising method performance. Finally (section 5), we apply our proposal to a number of realistic problem-solving methods, namely for hierarchical classification, parametric design (methods from XCON and VT), and consistency-based diagnosis (the GDE-method).

2 What is anytime reasoning?

Anytime algorithms are defined as algorithms that return some answer for any allocation of computation time, and are expected to return better answers when given more time [BD89]. This is in contrast with traditional algorithms which guarantee a correct output only after termination, and no guarantees are given for any intermediate results. The behaviour of an anytime algorithm is described by a *performance profile*. A performance profile describes how the quality of the output of the algorithm varies as a function of the computation time. The quality measure of the output may be any characteristic of the result of an algorithm that we find significant. One anytime algorithm could have several performance profiles tracking different attributes of the results it returns. A performance profile is typically given in the form of a graph that plots output quality against runtime.

It is clear that anytime behaviour of PSMs is desirable. Such PSMs are usable even when there is insufficient time to compute complete solutions. This is often the case given the intractable nature of typical tasks for PSMs. Also, such PSMs are applicable in real-time situations when the available computation time is often short and not known in advance. Thirdly, they offer the user the possibility to trade solution-quality against computation time, making the PSMs more widely applicable when selected from a library.

Perhaps surprisingly, anytime PSMs occur frequently. Many PSMs in the literature turn out to have an anytime nature, even when they were not developed with this purpose in mind. We have analysed the PSMs from a modern textbook on knowledge-based systems [Ste95], and have found that many of the methods discussed there have anytime behaviour. This will be illustrated in section 5, where we discuss examples which are all taken from this textbook, many of which are used in realistic KBS applications.

The study of anytime algorithms is fairly recent, and started with the introduction of the notions of anytime algorithm and performance profile by [DB88]. Subsequently, work has been done on combining and compiling anytime components from libraries using performance profiles (e.g. [ZR96]). Also, work has been done by a variety of people on special purpose anytime algorithms for various application areas (e.g planning [BD89; DB88], diagnosis [Pos93], search ([Kor90]) and scheduling [BD94]).

Boddy [Bod91] identifies a number of families of algorithms which often have anytime algorithms: numerical approximation, heuristic search, probabilistic algorithms (eg Monte Carlo methods), probabilistic inference (eg belief networks), and discrete symbolic processing. We deal with algorithms from this last family. These algorithms often add or remove elements to finite sets representing an approximate answer, and gradually reduce the difference between that set and a set representing the correct answer.

[Zil96] gives a number of desirable properties of anytime algorithms:

1. *Interuptability:* the algorithm can be stopped at anytime and provide some answer.
2. *Monotonicity:* the quality of the result is a non-decreasing function of the computation time.
3. *Measurable quality*: the quality of an approximate result can be determined precisely.
4. *Diminishing returns:* the improvement in solution

[3] i.e. the functional I/O relation

quality is largest at the early stages of computation, and it diminishes over time.

5. *Consistency:* for a given amount of computation time on a given input, the quality of the result is always the same.
6. *Recognisable quality*: the quality of an approximate result can be easily determined at run-time.
7. *Preemptability:* the algorithm can be suspended and resumed with minimal overhead.

Notice that the first two properties constitute the definition of anytime algorithms as given at the start of this section, and are therefore required, rather than desirable. The third and fourth property are also applicable to the anytime descriptions of PSMs that we propose in this paper. The fifth property (consistency) is also (but trivially) applicable since we only deal with deterministic computation. Only the last two properties are not applicable to our work for the following reasons: Recognisable quality is not applicable since we are not dealing with dynamic monitoring of algorithms, and therefore this property is not relevant. Preemptability is not applicable, since in our analysis algorithms are only stopped, and never resumed.

3 Describing gradual properties of PSMs

The motivating question for this paper as given in the introduction was: *how should the performance of a PSM be described so that it can be used as an applicability condition?*

Instead of the empirically obtained quality-performance graphs used for this purpose in the literature, we aim for an analytic treatment of the performance profiles in the form of an axiomatic description. This has the advantages of not needing expensive and unreliable empirical performance observations, and of giving more insight in the actual behaviour of the PSM. It also fits better with existing approaches of describing applicability condition in the Knowledge Engineering literature.

The second Zilberstein property above states that the output quality is a gradual property of the available computation time. In [vHtT98], we have proposed a general framework for describing gradual properties of the output of PSMs as a function of gradual properties of their input. If we regard computation time as a special "input parameter" to the PSM, anytime PSMs become a special case of what can be described in our proposed framework for gradual properties of PSMs.

In this section we briefly summarise our framework for describing gradual properties of PSMs. In the next sections, we will use this framework for the description of performance profiles of PSMs.

The framework for gradual properties of PSMs that is described in [vHtT98] is based on a traditional pre- and postcondition description of PSMs: given that the preconditions hold on the input of a PSM, then after termination, the postconditions are guaranteed to hold on the output of the PSM (ie. the functionality of the PSM is achieved). Unlike traditional pre/postcondition frameworks however, our conditions are gradual. To be more precise, our pre- and postconditions each have an additional parameter. Partial orderings are defined on these parameters that describe the degree to which the conditions are fulfilled, with the minimal element signifying completely fulfilled conditions. The two most important proof obligations in this framework for the description of anytime PSMs are as follows[4]:

- a first proof obligation is to show how a change in fulfillment of the preconditions causes a change in fulfillment of the postconditions;
- A second proof obligation shows that completely fulfilled preconditions imply completely fulfilled postconditions.

In order to apply this framework to anytime PSMs, we must decide on the partial orderings defining gradual fulfillment of pre- and postconditions. For the ordering on the preconditions, we obviously choose the amount of runtime available to the PSM. Because the minimal element of this ordering must correspond with completely fulfilled preconditions, we choose the maximally required runtime as the minimal element, and larger elements under the ordering correspond with ever less computation time(!). Because runtime cannot be less then 0, the elements range from complete runtime (the minimal element) down to 0.

As the ordering on the postconditions, we take whatever quality measure is taken as characteristic of the result of the algorithm. The second Zilberstein property guarantees that a suitable ordering can be defined on this characteristic. The third Zilberstein property demands that this must be a measurable value.

As mentioned before, a performance profile plots output quality against runtime. The two axes of such a graph now correspond exactly with our two measures: the precondition measure (runtime) as x-axis, and the postcondition measure (output quality) as y-axis (see the figures later in this paper).

4 Guidelines for anytime descriptions

Describing applicability criteria for PSMs is one of the hardest tasks when building a PSM library. These criteria must accurately capture both the preconditions and the functionality of the PSMs in the library. The accuracy of

[4] [vHtT98] defines three more proof obligations, but these are less relevant when applying the framework to describe anytime behaviour: one proof obligation corresponds exactly with Zilbersteins second property (and therefore holds by definition for anytime algorithms); a second obligation concerned the relation between gradual and completely fulfilled preconditions, which in the current case simply means that after enough runtime, the complete solution is computed; finally, the usual correctness property must be shown for the PSM but this time with respect to the gradual versions of the pre- and postconditions.

these descriptions is crucial for the later ability to retrieve the appropriate elements from the library.

This task is already hard in current PSM libraries where the applicability conditions only have the form of "labels", intended to be understood by human users, but without any further formal semantics. The task becomes even harder when applicability conditions take the form of complex expressions in a formal language suited for automatic manipulation (as proposed in e.g. [BPG96]).

Our proposal to use performance profiles of PSMs threatens to make this task even harder: in our own studies we have found that formulating gradual pre- and postconditions is even harder than formulating traditional pre- and postconditions, since the gradual versions require an even more detailed analysis of the behaviour of the PSM than is already required for traditional applicability criteria.

The hardest steps in describing gradual pre- and postconditions are: (1) defining a suitable ordering on the precondition; (2) defining a suitable ordering on the postconditions; and perhaps most difficult of all (3) defining the actual gradual functionality of the PSM (ie its gradual postconditions) given some gradual pre-conditions.

The good news is that when applying our general framework to anytime performance we can make some of these choices in advance (so they don't have to made by the library builder), and we can give a structured schema for some of the descriptions that must be supplied by the library builder.

The choices concerning the preconditions disappear altogether: We always simply add the available runtime as an additional parameter, and apply the obvious ordering to this parameter [5].

What remains is the formulation of the postconditions (ie. the gradual functionality of the PSM as a function of increasing runtime). *We have found that this formulation can always be structured in the same way*. To describe the anytime functionality, four axioms are needed, each of which describes a different aspect of the anytime behaviour, as follows:

Initial behaviour: The initial period during which the behaviour of the method is constant. Many anytime algorithms start producing some output immediately, but the example in section 5.3 shows that some methods need an initial "startup period" before they start producing intermediate output.

Growth direction: This is the direction in which the quality of the intermediate output changes with increasing runtime. The second Zilbertstein property (monotonicity) guarantees that quality increases, and this axiom states what is meant by "increase".

[5] While noting that the minimal element corresponds with the maximum runtime, as explained in the previous section

Growth rate: The amount of increase in quality at each step during the computation. This increase in quality can be constant at each step, but may also vary during the computation (as stated in the fourth Zilberstein property: diminishing returns).

End condition: The amount of runtime needed for the method to achieve its full (ie. traditional) functionality. After this point, the quality of the output no longer increases, since the maximum quality has been achieved.

In [GtTvH99] we have put forward the hypothesis that this scheme of four axioms would be suitable to describe a wide class of anytime behaviours. One of the results of this paper is the confirmation of this hypothesis, by showing that this scheme can be used to describe the anytime behaviour of a number of different and realistic PSMs from the literature.

5 Example anytime descriptions of PSMs

In this section we will apply the above scheme for describing performance profiles of PSMs to a number of concrete PSMs. These methods are all described in a modern KBS textbook [Ste95; Part III].

First we discuss three methods for a classification task. The first two examples (linear candidate confirmation and linear candidate confirmation with forward filtering) are theoretical and simple, and are meant to introduce our proposal. The third (hierarchical classification) is more realistic, and similar to the method used in the MDX system for diagnosing liver diseases [CM83]. We will then turn to the task of parametric design, and discuss the methods used in the XCON system (constraint clustering) [McD82] and in the VT systems (propose and revise) [MSM88]. Finally, we discuss the GDE method for the task of consistency-based diagnosis [Rei87; dKW87].

5.1 The classification task

In a classification task, we are given a set of candidate classes and a set of observed properties of a particular individual, and we must compute which candidate classes satisfy the classification criterion on the given properties. The details of the classification criterion can vary, and are not relevant to our discussion (see [Ste95; Ch. 7] for the definition of various classification criteria such as candidates which explain, match or cover the given observations).

Slightly more formally, the task CLASSIFICATION has as inputs a set of candidate classes *Cs* and a set of observations *Obs*, and must compute all classes from *Cs* that satisfy the classification criterion on *Obs*:

$$\text{CLASSIFICATION}(Cs, Obs) = \{C_i | C_i \in Cs \wedge criterion(C_i, Obs)\}$$

5.2 Linear candidate confirmation (MC1)

A trivial PSM for the classification task is to iterate over all candidate classes, and add them to the output if they satisfy the classification criterion[6,7]:

```
MC1( n, Cs,Obs):
  output = ∅
  candidates = {Ci | Ci ∈ Cs  ∧ i ≤ n }
  for Ci ∈ candidates
  do if criterion(Ci,Obs)
     then output = output+Ci
  done
  return output
```

The algorithm MC1 is as given without the additional boxed text. If the boxed text is added to the code, we obtain an anytime version of MC1 that we will indicate with $MC1_a$[8]. As mentioned above, the additional parameter n signifies the available runtime (ie the algorithm terminates after n steps), and is used as the ordering on the preconditions. As ordering on the postconditions (= quality measure of the output) we will use the subset ordering $\subseteq$ on the output set.

Following the guidelines from the previous section, the gradual functionality of $MC1_a$ can now be specified as follows:

MC1-initial: Initially (with zero runtime) no solutions are computed:

$$MC1_a(0,Cs,Obs) = \emptyset$$

MC1-direction: The solution set only grows (and never decreases):

$$MC1_a(n,Cs,Obs) \subseteq MC1_a(n',Cs,Obs)$$

MC1-rate: Each additional computation step adds at most one solution[9]:

$$|MC1_a(n+1,Cs,Obs)| - |MC1_a(n,Cs,Obs)| \leq 1$$

MC1-end: After considering all candidates, we have obtained the full functionality:

$$n \geq |Cs| \rightarrow MC1_a(n,Cs,Obs) = MC1(Cs,Obs)$$

Assuming a uniform distribution of the solutions over the candidate set, the theoretically derived performance profile determined by these axioms is shown in figure 1a[10].

[6] This method is called MC1 in [Ste95; Ch. 7].

[7] The languages used for both program code and specifying axioms are close to those used in the KIV interactive program verifier. We expect that all the formal definitions in this paper can be easily transcribed in the KIV system [Rei95]. It should then be possible to provide machine-verified proofs for all the result in this paper.

[8] This same box-notation is used for the other algorithms in this paper.

[9] the notation $|S|$ is used for the size of a set S

[10] Our graphs are plotted as continuous functions, but in reality the increases in output quality are stepwise.

5.3 Linear confirmation with forward filtering (MC2)

This method is equal to MC1, but first applies an initial forward reasoning step, in which some of the observations are used to filter the set of possible candidates. The method MC1 is applied to the resulting candidate set. An assumption of this method is that the additional cost of the filter step is outweighed by the reduction in cost of applying the linear confirmation step to a smaller candidate set. Furthermore, it assumes that the forward filtering step only removes candidates which are not solutions to the classification problem (ie the filtering step is sound). Instead of a single set of observations, MC2 receives two sets of observations as input, one to be used in the forward filtering step, the other to be used in the candidate confirmation step:

```
MC2( n, Cs,Obs1,Obs2):
  output = ∅
  candidates = {Ci | Ci ∈filter(Cs,Obs1)
                        ∧ i ≤ n }
  for Ci ∈ candidates
  do if criterion(Ci,Obs2)
     then output = output+Ci
  done
  return output
```

Following the guidelines from the previous section, the gradual functionality of $MC2_a$ can now be specified as follows:

MC2-initial: Unlike MC1, MC2 does not immediate produce intermediate solutions, since it first completes the forward filtering step. If n_f indicates the duration of the filtering step, then:

$$n < n_f \rightarrow MC2_a(n,Cs,Obs_1,Obs_2) = \emptyset$$

MC2-direction: similar to [MC1-direction].

MC2-rate: similar to [MC1-rate].

MC2-end: The total required runtime of MC2 is the sum of the two stages. The duration of the filtering step is n_f, and the duration of the confirmation stage is determined by the number of candidates that remain after the filtering step:

$$n \geq |filter(Cs,Obs_1)| + n_f \rightarrow$$
$$MC2_a(n,Cs,Obs_1,Obs_2) = MC2(Cs,Obs_1,Obs_2)$$

The performance profile determined by these axioms is shown in figure 1b. It shows a constant increase in quality (similar to MC1), but only after an initial period needed for the filtering step.

The assumption that the costs of the filter step must be outweighed by the savings can now also be stated more precisely. The costs of the filtering step is n_f, the savings equal the reduction in the candidate set: $|Cs| - |filter(Cs,Obs)|$.

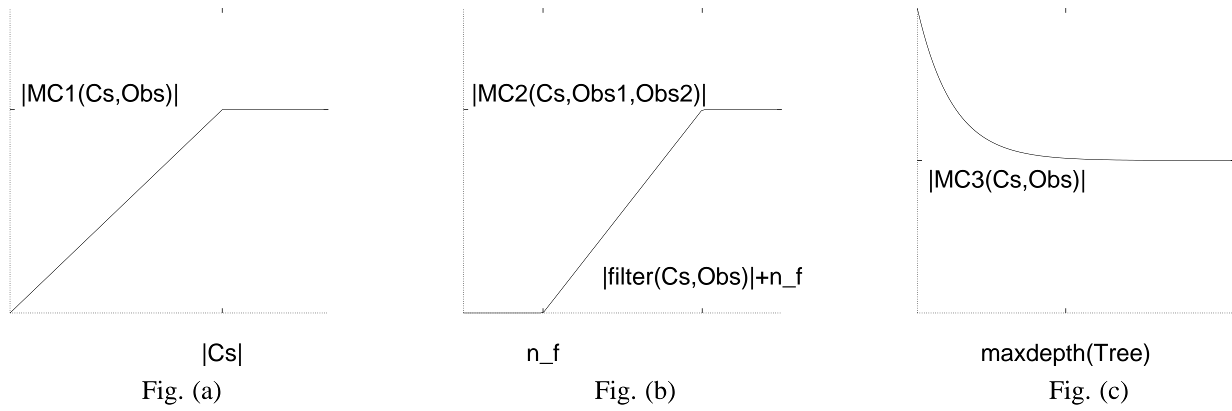


Figure 1: Performance profile of MC1, MC2 and MC3.

The assumption holds when $n_f < |Cs| - |filter(Cs,Obs)|$. This can also be written as $n_f + |filter(Cs,Obs)| < |Cs|$. Now notice that this states precisely that the end time of MC2 is less then the end time of MC1 (see axioms [MC1-end] and [MC2-end]).

5.4 Hierarchical classification (MC3)

The first realistic PSM that we will discuss is hierarchical classification (used among others in the MDX system [CM83]). This method no longer does a linear traversal of the candidate set. Instead, the candidate set is organised as the leaves of a tree. The nodes in this tree are “abstract classes”, representing abstractions of sets of candidates. The PSM recursively descends down the tree, at each level deciding if the abstract classes still satisfy the observations. If yes, the method continues to descend down that part of the tree, if no, the entire tree below the abstract class is pruned. At each step of the PSM, the intermediate solution is the set of all candidates (= all leaves) that can be found under the currently considered abstract classes.

```
MC3(n, Tree,Obs):
   output = leaves(Tree)
   current = {Tree}
   next = []
   d=1
   while current ≠ ∅ ∧ d ≤ n
   do for c ∈ current
      do if   not criterion(c,Obs)
         then output = output - leaves(c)
         else next = next + children(c)
      done
      current = next
      next = ∅
      d = d+1
   done
   return output
```

The gradual functionality of MC3 can be characterised as follows:

MC3-initial: Initially, all candidates are still potential solutions:

$$MC3_a(0, Tree, Obs) = leaves(Tree)$$

MC3-direction: an extra computation step can only decrease the set of potential solutions:

$$MC3_a(n+1, Tree, Obs) \subseteq MC3_a(n, Tree, Obs)$$

MC3-rate: Taking b as the branching factor of the tree, and assuming that at each level of the tree, at least 1 of the abstract classes satisfies the criterion, and assuming a balanced tree, then each step reduces the candidate set by a factor b:

$$|MC3_a(n+1, Tree, Obs)| \geq \frac{|MC3_a(n, Tree, Obs)|}{b}$$

MC3-end: MC3 needs as many steps as there are levels in the tree

$$n \geq maxdepth(Tree) \rightarrow$$
$$MC3_a(n, Tree, Obs) = MC3(Tree, Obs)$$

Notice that for candidate classes of exponential size (which occur for multi-class classification), both MC1 and MC2 require exponential runtime (since they perform a linear traversal of the exponential candidate set), while MC3 only requires linear runtime (namely the depth of the tree, axiom [MC3-end]). This is all consistent with known results about the complexity of hierarchical classification [GSC87].

Using the performance profiles

We have now defined anytime performance profiles for three different classification methods. We give three examples how these profiles help us with selecting methods from a library. First, the profiles tell us how we can trade computation time for solution quality. For example, fig 1c shows

that the increase in quality of MC3 (ie the decrease in the candidate set) is exponential while for MC1 and MC2 this linear. If it is important to quickly obtain a good approximation of the final solution, then MC3 is more attractive than MC1 or MC2.

Secondly, we see from the performance profiles that MC1 and MC2 are incomplete (but sound) approximations of the final solution. The intermediate solutions of MC3 on the other hand are unsound approximations of the final solution, since the profile of MC3 approaches the solutions from above, rather than from below.

Thirdly, we see that not all methods start to produce approximate solutions immediately. If such a property is important (e.g. in a setting where some solution is always required but no guarantees can be given on the available runtime), then MC2 is unattractive.

5.5 The parametric design task

We now turn to a very different type of task, namely the task of parametric design. In this task we are given a set of parameters and a set of constraints, and have to compute an assignment for each parameter such that these assignments are consistent with the given constraints.

Slightly more formally:

$$\text{PARAMETRIC-DESIGN}(Ps, Cs) = S, \text{with}$$
$$S = \{(P_i, V_i) | P_i \in Ps\} \wedge consistent(Cs, S).$$

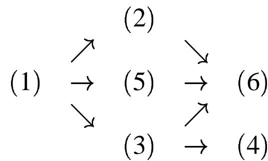
In the next two sections, we give two problem solving methods for performing this parametric design task.

5.6 Param. design by constraint clustering (XCON)

XCON [McD82] is a method for parametric design based on a particular organisation of constraints. The constraints are divided in clusters. The constrains within a cluster have much mutual dependencies, but no dependencies exist with constraints in other clusters. This makes it possible to solve the problem in separate steps, with backtracking occuring only within each step (namely per cluster), and not between steps. An example of such steps in the XCON application (configuring VAX mainframe computers for Digital) are:

1. make the order complete
2. configure the set of components for the CPU cabinet(s)
3. configure the set of components for the Unibus cabinet(s)
4. configure the panels for the Unibus cabinet(s)
5. configure the spatial layout of the cabinets
6. configure the cabling

These are ordered as follows:



This partial ordering can be used to determine an execution order of the steps. The inputs of the XCON method are a list of constraint clusters and the set of parameters. The order in the list of clusters is assumed to respect the partial dependency-ordering among the clusters.

The XCON method simply iterates over the constraints clusters, solves each cluster separately (no details for this step are given in the definition below), and adds the assignments found for each step to the current output:

```
XCON( n, Ps,[Cs1,...,Csk]):
   output = ∅
   for i=1 to min(n, k )
   do CurrentPs = relevant(Ps,Csi)
      S = {(Pi,Vi) | Pi ∈ CurrentPs}
          ∧ consistent(Cs,S)
      output = output + S
   done
   return output
```

The gradual functionality of XCON can be characterised as follows:

XCON-initial: Initially no assignments have been computed.

$$XCON_a(0, Ps, [Cs_1, ...]) = \emptyset$$

XCON-direction: The set of assignments that is computed grows monotonically.

$$XCON_a(n, Ps, [Cs_1, ...]) \subset$$
$$XCON_a(n+1, Ps, [Cs_1, ..])$$

XCON-rate: The maximal number of new assignments for a cluster is the number of relevant variables of the constraints in that cluster. This is a maximum, since some parameters might have been computed in earlier steps (clusters). Because of the assumption of independency between steps, such assignments never violate the constraints of the current step.

$$|XCON_a(n+1, Ps, [Cs_1, ..])| -$$
$$|XCON_a(n, Ps, [Cs_1, ..])| \leq |\text{relevant}(Ps, Cs_{n+1})$$

XCON-end: The complete functionality is obtained after k steps, with k the number of constraint clusters in the input.

$$n \geq k \rightarrow \quad XCON_a(n, Ps, [Cs_1, .., Cs_k]) =$$
$$XCON(Ps, [Cs_1, .., Cs_k])$$

These axioms determine the performance profile, as shown in figure 2a. The graph shows a stepwise increase of the output quality in k steps.

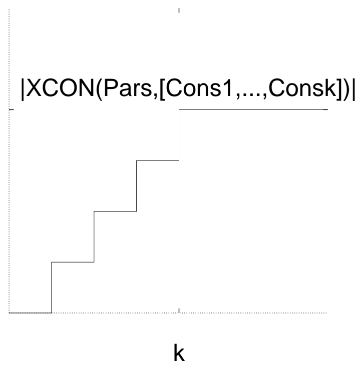


Fig (a)

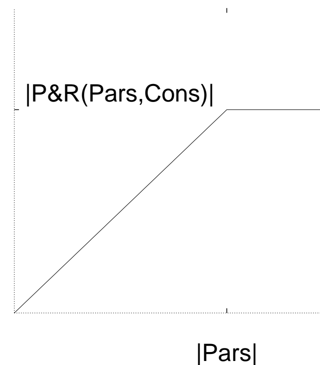


Fig (b)

Figure 2: Performance profile of XCON and P&R

5.7 Parametric design by propose and revise (P&R)

Another well known method to obtain the functionality of parametric design is Propose & Revise (P&R). The P&R method iterates over the set of parameters. In each step, P&R takes a new parameter and proposes a likely value for that parameter. This new assignment becomes part of the current partial design. If this partial design is consistent with the constraints, a next step can be taken. If the partial design violates the constraints then the partial design will be revised such that it becomes consistent with the constraints. After fixing the partial design the process continues with the next parameter.

```
P&R( n, Ps,Cs):
   output = ∅
   for i=1 to  min(n, |Ps| )
   do Vi = propose(output,Pi)
      output = output = (Pi,Vi)
      if ¬consistent(Cs,output)
      then output = revise(Cs,output)
   done
   return output
```

This method uses domain knowledge in the propose step and in the revise step, whereas the XCON method uses domain knowledge in the way the constraints are clustered.

For XCON we used the set of assignments as the quality measure for the computation. This same measure cannot be used for P&R. Unlike XCON, P&R assignments in earlier steps might have to be revised in later steps. As a result, the set of assignments does not grow monotonically, violating the second Zilberstein property. For this reason, we use another quality measure for P&R, namely the set of assigned parameters instead of the set of assignments (ie parameters plus their values). This set does grow monotonically, since parameters might be revised, but once assigned, a parameter is never left without a value in later stages of the compuation.

P&R-initial: Initially no assignments have been computed.

$$P\&R_a(0,Ps,Cs) = \emptyset$$

P&R-direction: Unlike XCON, the set of assignments does not grow monotonically in P&R, but the set of assigned parameters does (assigned parameters might be revised). The set $\{P_i|(P_i,V_i) \in P\&R_a(n,Ps,Cs)\}$ contains all parameters that have been assigned a value after n steps. The monotonic growth is then:

$$\{P_i|(P_i,V_i) \in P\&R_a(n,Ps,Cs)\} \subseteq \{P_i|(P_i,V_i') \in P\&R_a(n+1,Ps,Cs)\}$$

P&R-rate: P&R iterates over the set of parameters, therefore each step yields exactly one additional assigned parameter:

$$|\{P_i|(P_i,V_i) \in P\&R_a(n+1,Ps,Cs)\}| = |\{P_i|(P_i,V_i) \in P\&R_a(n,Ps,Cs)\}| + 1$$

P&R-end: The method needs as many steps as there are parameters:

$$n \geq |Ps| \rightarrow P\&R_a(n,Ps,Cs) = P\&R(Ps,Cs)$$

These axioms determine the performance profile, as shown in figure 2b. The set of assigned parameters grows at a constant rate during the computation.

Concerning axiom [P&R-direction]: This axiom allows that partial assignments are not a subset of the final assignments (ie partial assignments are allowed to be unsound). However, in the VT application which used the P&R method, it turns out that the domain knowledge used in the propose step is so good that revision is almost never needed: on test-cases with 1000-2000 parameters (and therefore as many propose steps), there were only some 10-20 violations (and therefore as many revision steps), ie only 1% of the proposed values were wrong [MSM88]. Thus, although the soundness of XCON's approximations (expressed by XCON's axiom [XCON-direction]) cannot be guaranteed for P&R, in practice P&R comes very close.

Using the performance profiles

Again, the performance profiles for the different methods can be used for selecting the methods from a library. It follows from axioms [XCON-end] and [P&R-end] that P&R divides the process in $|Ps|$ steps, and XCON in k steps (k the number of constraint clusters). Since every cluster will assign at least one additional parameter, we have $|Ps| \geq k$, so P&R divides the process in much smaller steps then XCON. If it is important that new solutions are produced at a constant rate during the computation process (e.g. because of interface requirements with users or other programs), then P&R is more attractive from an anytime perspective.

5.8 The consistency based diagnosis task

In consistency-based diagnoses [Rei87] we are given a theory describing the intended behaviour of a system (called the system description, *SD*), and some observations of the systems actual behaviour, *Obs*. A diagnosis problem exists when *Obs* is inconsistent with *SD* (ie. the system is observed not to function as intended). The goal is to compute a minimal set of components *Diag* which can "explain" the abnormal behaviour, ie. assuming that these components are abnormal suffices to make *Obs* again consistent with *SD*.

$$DIAGNOSIS(SD, Obs) = Diag \quad \text{such that } consistent(SD \cup Obs, Diag)$$

5.9 The GDE method

The best known PSM for this method is the GDE method [dKW87]: It first computes so called conflict-sets. A conflict-set is a set of components which, given the observations, can not all be correct, ie at least one component in each conflict set must be part of the diagnosis. After having computed all minimal conflict-sets, GDE then uses these conflict-sets to compute so called hitting-sets. A hitting-set is a set of components that contains at least one element from every conflict-set. It follows that each minimal hitting-set is a diagnosis.

Pos [Pos93] has shown how this can be turned into an anytime algorithm, namely by only computing the minimal conflict-sets sets up to a maximum size n instead of computing *all* minimal conflict-sets, and then computing all minimal hitting sets for this limited collection of conflict-sets. This anytime version of GDE is as follows:

```
GDE( n, SD,Obs):
Cs = all minimal conflict-sets C
      with |C| ≤ n
output = all minimal hitting sets for Cs
return output
```

The gradual functionality of GDE can be characterised as follows:

GDE-initial: Initially no diagnoses are computed.

$$GDE_a(0, SD, Obs) = \emptyset$$

GDE-direction: For every diagnosis from step n, there will be a superset diagnosis in step $n+1$, in other words: individual diagnoses grow monotonically during the computation:

$$D \in GDE_a(n, SD, Obs) \rightarrow \exists D' : D \subseteq D' \wedge D' \in GDE_a(n+1, SD, Obs)$$

GDE-rate: Unfortunately, we have not been able to estimate by how much individual diagnoses grow when the maximum size of the conflict sets increases by 1. Thus, in the following expression, we have no value for m.

$$D \in GDE_a(n, SD, Obs) \wedge D' \in GDE_a(n+1, SD, Obs) \wedge D \subset D' \rightarrow |D'| - |D| \geq m$$

GDE-end: Again unknown. We have not been able to give a reasonably sharp upperbound on the maximum size of the conflict sets that is required to compute all diagnoses. Of course, after some value k, the algorithm will compute no more diagnoses (since all have been computed), but we do not know the value of k:

$$n \geq k \rightarrow GDE_a(n, SD, OBS) = GDE(SD, Obs)$$

Because of the unknown parameters in axioms [GDE-rate] and [GDE-end], we are not able to give a graphical rendition of the performance profile for the GDE method.

6 Conclusions & Future Work

Conclusions In the KE literature, libraries of PSMs have been indexed using functional descriptions and knowledge requirements. In this paper, we have proposed the use of non-functional properties (in our case anytime performance profiles) as a library index for PSMs.

We have given axiomatic descriptions of anytime behaviour of PSMs. This is unlike existing work on anytime algorithms, which obtains performance profiles by simulation and measurement. Such empirically obtained profiles are dependent on the quality of the simulations, which are often expensive, and also not very reliable since they depend on the particular input distribution used for the simulations. On the other hand, our axiomatic descriptions are often limited to giving an upper- or lower-bound on the rate of the quality improvement, whereas empirical performance profiles do obtain values for the improvement rate.

In order to make it easier for library builders to give actual performance profiles for the PSMs in their libraries, we

have given guidelines for constructing such an axiomatic descriptions: each description should consist of four statements, describing initial behaviour of the PSM, direction of quality change, rate of quality of change per time unit and the time at which the optimal output quality is obtained. This regularity in the description of the dynamic behaviour of PSMs confirms a hypothesis put forward in our earlier work [GtTvH99].

Our axiomatic description of performance profiles is based on our earlier and more general proposal for describing gradual properties of PSMs [vHtT98]. It turns out that performance profiles can be described in our framework for describing gradual properties. This was not obvious beforehand because the framework must be used in a slightly non-standard way. It was designed to deal with functional properties (I/O-pre/post-conditions), while anytime behaviour is a non-functional property (concerning also the computation time, and not only the I/O relation).

Future Work In the axioms in this paper, we give only upper- or lowerbounds for the rate of quality improvement (the third axiom in our general scheme), in particular when these rates are based on the quality of heuristic knowledge. Instead of such upper- and lowerbounds, we would like to give more precise *expected values* for the improvement rate, based on the quality of the heuristic. For example, in MC1 the solution set increases with at most one element each computation step. However, it is easy to see that assuming a uniform distribution of solutions over candidate set, the expected rate of increase one element for each k steps (k the number of candidates divided by the number of solutions). Furthermore, using a heuristic ordering of the candidate set, we can expect one additional element for each $k(n)$ steps, with $k(n)$ a increasing function of n. We have currently no formal framework in which to express these and similar statements.

We would also like to study which anytime behaviour is more attractive under which circumstances. For example, it is clear that dividing a given runtime in a large number of small steps is more attractive than dividing it in a small number of large steps (compare XCON, fig 2a and P&R, fig 2b). Less clear is the question whether the behaviour of MC2 is more attractive than MC1 (because MC2 uses less total runtime) or whether MC1 is more attractive than MC2 (because its quality increase is more evenly divided over the computation time). To our knowledge, the literature on anytime algorithms has not tackled this question until now.

Finally, in [GtTvH99] we have developed some simple techniques for proving dynamic properties of KBS, and we used the interactive theorem prover KIV [Rei95] to verify a simple PSM (in fact, MC1). All the formal definitions in this paper (both program code and axioms) have been given in a syntax already very close to that used in the KIV systems. We expect that our techniques can be applied to verify the axiomatic anytime descriptions given in this paper, yielding machine-assisted formal proofs of the anytime behaviour of realistic PSMs.

References

[BD89] M. Boddy and T. Dean. Solving time-dependent planning problems. In *Proceedings IJCAI–89*, Detroit, Michigan USA, August 1989.

[BD94] M. Boddy and T. Dean. Deliberation scheduling for problem solving in time-constrained environments. *Artificial Intelligence*, 67:245–285, 1994.

[Bod91] M. Boddy. Anytime problem solving using dynamic programming. In *Proceedings of the ninth National conference on artificial intelligence AAAI–91*, pages 738–743, 1991.

[BPG96] V. R. Benjamins and C. Pierret-Golbreich. Assumptions of problem-solving methods. In N. Shadbolt, K. O'Hara, and G. Schreiber, editors, *Lecture Notes in Artificial Intelligence, 1076, 9th European Knowledge Acquisition Workshop, EKAW-96*, pages 1–16, Berlin, 1996. Springer-Verlag.

[CM83] B. Chandrasekaran and S. Mittal. Deep versus compiled knowledge approaches to diagnostic problem solving. *International Journal of Man-Machine Studies*, 19:425–436, 1983.

[DB88] T. Dean and M. Boddy. An analysis of time-dependent planning. In *Proceedings of the seventh National conference on artificial intelligence AAAI–88*, pages 49–54, Saint Paul, Minnesota, 1988.

[dKW87] J. H. de Kleer and B. C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32:97–130, 1987.

[FB98] D. Fensel and R. Benjamins. The role of assumptions in knowledge engineering. *International Journal on Intelligent Systems (IJIS)*, 13(8):715–748, 1998.

[FS98] D. Fensel and R. Straatman. The essense of problem-solving methods: Making assumptions for gaining efficiency. *International Journal of Human-Computer Studies (IJHCS)*, 48(2):181–215, 1998.

[GSC87] A. Goel, N. Soundarajan, and B. Chandrasekaran. Complexity in classificatory reasoning. In *AAAI–87*, pages 421–425, 1987.

[GtTvH99] P. Groot, A. ten Teije, and F. van Harmelen. Formally verifying dynamic properties of knowledge based systems. In *Proceedings 11th European Workshop on Knowledge Acquisition, Modeling, and Management (EKAW '99)*, Lecture Notes in Artificial Intelligence. Springer Verlag, 1999.

[Kor90] R. Korf. Real-time heuristic search. *Artificial Intelligence*, 42(2):189–212, 1990.

[McD82] J. McDermott. R1: A rule-based configurer of computer systems. *Artificial Intelligence*, 19:39–88, 1982.

[MSM88] S. Marcus, J. Stout, and J. McDermott. VT: An expert elevator designer that uses knowledge-based backtracking. *AI Magazine*, Spring:95–111, 1988.

[Pos93] A. Pos. Time-constrained model-based diagnosis. Technical report, University of Twente, department of computer science, 1993. Master's thesis.

[Rei87] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32:57–96, 1987.

[Rei95] W. Reif. The KIV-approach to Software Verification. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software – Final Report*. Springer LNCS 1009, 1995.

[SB95] R. Straatman and P. Beys. A performance model for knowledge-based systems. In *Proceedings of European Symposium on the Validation and Verification of Knowledge Based Systems (EUROVAV'95)*, pages 253–263, Adeiras, June 1995. Université de Savoie, Chambér.

[Ste95] M. Stefik. *Introduction to Knowledge-Based Systems*. Morgan Kaufmann, 1995.

[vHtT98] F. van Harmelen and A. ten Teije. Characterising approximate problem-solving by partial pre- and postconditions. In *Proceedings of ECAI'98*, pages 78–82, Brighton, August 1998.

[WAS98] B. Wielinga, J. Akkermans and A. Schreiber. A competence theory approach to problem solving method construction. *Internat. Journal of Human-Computer Studies, special issue on problem-solving methods*, 49(4), 1998.

[Zil96] S. Zilberstein. Using anytime algorithms in intelligent systems. *Artificial Intelligence Magazine*, fall:73–83, 1996.

[ZR96] S. Zilberstein and S.J. Russell. Optimal composition of real-time systems. *Artificial Intelligence*, 82(1–2):181–213, 1996.