Lecture Notes in Artificial Intelligence, 1952

Handling Cases and the Coverage in a Limited Quantity of Memory for Case-Based Planning Systems*

Flavio Tonidandel [1] and Márcio Rillo [1,2]

[1] Universidade de São Paulo - Escola Politécnica
Av. Luciano Gualberto 158, trav3 São Paulo - SP - Brazil - 05508-900
[2] Faculdade de Engenharia Industrial
Av. Humberto de A. Castelo Branco, 3972 - São Bernardo do C. - SP - Brazil - 09850-901
e-mails: flavio@clarke.lac.usp.br ; rillo@lsi.usp.br

Abstract. The majority of case-based planning systems consider an infinite case memory to store their cases. However, the size of the case memory is limited and it can become a barrier for case-based systems efficiency when it is full. This paper presents a method that refines and abstracts cases in order to release memory space for a new case. However, in some situations, some cases must be chosen to be deleted, and the method incorporates a case-deletion policy that achieves a lower bound for coverage depletion. Besides this paper can deal with a limited quantity of memory to store cases, the case-deletion policy also reaches better results for coverage-preserving than the case-addition policy proposed by Zhu and Yang [11].

1 Introduction

Case-based planning (CBP) is considered as a meaningful method that improves efficiency in planning systems by the use of earlier performed plans. However, the efficiency of this kind of planner depends on several features. One of these is the *swapping problem* [3]. The problem is to choose which and how many plans can be stored as cases in a case memory. It happens for the reason that not all cases can be stored because the case retriever can become a time expensive process. On the other hand, an enlarged case-base increases the coverage of the case-base, that is the range of problems that a case-base can solve after the retrieval and adaptation processes.

The majority of CBP systems, like MRL [4], CAPLAN [7], and others, do not concern about keeping and handling the case memory. In addition, they admit an infinite memory to store cases, which in some systems or domains is not possible.

The concern about creating, keeping and handling a case-base is known as case-base maintenance. Some researches in this area are concentrating on the *swapping problem* [6,8,11]. A deletion-based process, particularly after the Smyth and Keane's work [8], became an interesting way to deal with this problem while guards against the coverage depletion. The Smyth and Keane's work introduces a case-deletion policy that deletes cases from a case-base and keep the coverage as high as possible.

* This research is supported by FAPESP under contract 98/15835-9.

However, Zhu and Yang [11] showed that Smyth and Keane's case-deletion policy does not guarantee the coverage-preserving property. In addition, they proposed another kind of policy based on case-addition that guarantees a lower bound for the coverage resulted by their algorithm with respect to the optimal coverage.

However, both deletion and addition-based approaches can reach another barrier besides the *swapping problem*: the memory size. While they handle cases to keep the coverage as high as possible, they also consider a quantity of the case memory as much as necessary to store cases and to execute their methods.

In case-based planning, the cases have different sizes and even with a limited number of cases to store, they occupy different quantities of memory. However, the quantity of memory is always limited, and for some micro machines or complex domains, the memory size can become a barrier to implement any policy to control the coverage depletion. This paper presents a method that works with a limited size of memory and preserves the coverage by a pre-defined lower bound.

To achieve our aim, the Transaction Logic (TR) [2] is used to formalize cases and plans. It allows the abstraction and refinement processes definitions that control case sizes in memory and release memory space for the storage of a new case. In addition, when a deletion of a case is necessary, a suitable case-deletion policy provides a lower bound for the coverage remained, which, in a pre-defined situation, is better than that one proposed by Zhu and Yang's case-addition policy [11].

2 The Transaction Logic in Planning

The Transaction Logic (TR) is a logical formalism proposed by Bonner and Kifer [2]. The TR is an extension of the first-order logic, by the introduction of the serial conjunction operator (⊗), where $\alpha \otimes \beta$ means "first execute α, and then execute β".

The following notation is used to describe a transaction : $P, D_0,...,D_n \models \phi$, where ϕ is a transaction formula and P is a set of TR formulas called transaction base. Each D_i is a database state, that is a set of first-order formulas called *literal*. Intuitively, P is a set of transaction definitions, ϕ is an invocation of some of these transactions; $D_0,...,D_n$ is a sequence of databases that represents an updating made by ϕ. On the other hand, a situation of a query is not given by a sequence of databases, but by just one state. For example, $P,D_k \models qry(c)$, where c is true in D_k.

Tonidandel and Rillo [9] introduced definitions for plans, goals and cases with the use of the Transaction Logic. Let $\boldsymbol{A}$ be a set of actions, where each action is a TR formula that performs updates, the following definitions can be stated:

Definition 1: (Plan and empty plan) *A plan* $\delta = \alpha_1 \otimes ... \otimes \alpha_n$ *is a TR formula, where* $\alpha_i \in \boldsymbol{A}$*;* $1 \le i \le n$*. An empty plan* δ_0 *is a plan without any action* $\alpha \in \boldsymbol{A}$.

Definition 2: (pln instance) *A pln instance is a plan* δ *or an empty plan* δ_0.

Definition 3: (Goal) *A goal G is a TR formula and has the following structure:*
G: *pln* $\otimes$ ***Df*** *; where Df is a set of queries that represents the desirable final state.*

When the planner finds a desirable sequence of actions to substitute a *pln* instance in order to reach ***Df***, the plan can become a case to be stored for future uses. A case is a modified plan by the insertion of initial and final states features:

Definition 4: (Case) *A case* η *is a TR rule:*

$$\eta \leftarrow \mathbf{Wi} \otimes \alpha_1 \otimes \ldots \otimes \alpha_n \otimes \mathbf{Wf} \quad ; \text{where:}$$

- η *is a TR rule that represents a stored case.*
- $\alpha_i \in \mathbf{A}$*; 1* $\leq i \leq n$*, a plan defined by the planner that satisfies a proposed goal.*
- ***Wi*** *is a set of queries in TR that represents the precondition of the case.*
- ***Wf*** *is a set of queries in TR that represents the pos-condition of the case.*

Intuitively, *Wi* is a set of queries for those *literal* that are deleted by the plan and that must be in the initial state to permit the plan execution. Otherwise, *Wf* are those *literal* that are not in initial state and are inserted by the plan execution.

3 Case-Base Maintenance and Related Works

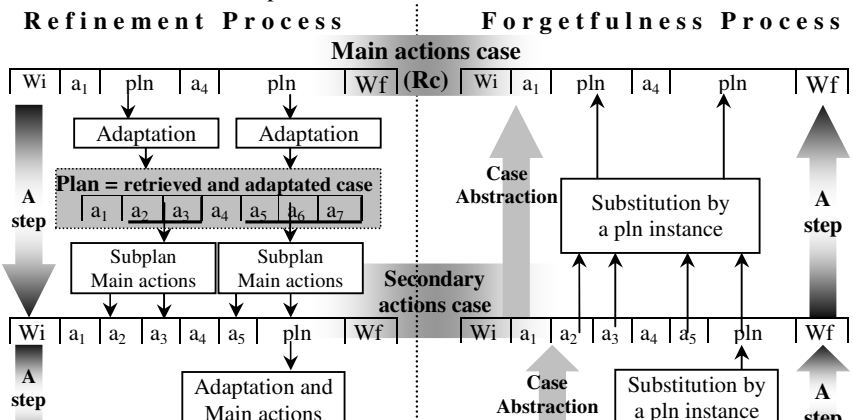
The majority of the researches in revising case-base contents are concentrated on the *swapping problem* [3]. The problem is to decide which cases will be deleted in order to prevent the retriever from becoming a time expensive process by an enlarged case-base.

Different strategies have been developed to deal with the *swapping problem.* One of these is the selective deletion that can be made by techniques involving the utility of each case [6] or the overall coverage, also named competence, of a case-base [8]. In this way, the work of Smyth and Keane introduces a case-deletion policy through a specification of cases by their coverage and reachability. They used an algorithm that can hierarchically choose which cases can be deleted in order to keep the overall coverage as high as possible

Recently, Zhu and Yang showed that Smyth and Keane's work cannot guarantee the coverage preservation property and proposed a selective utilization policy based on case-addition [11]. Instead of considering a retrieval time bounding to choose cases as in [10], they considered the coverage of cases to choose a limited number of them. With the choices performed by their algorithm, they reached a lower bound for the remained coverage that is 63% of the optimal coverage.

However, we do not concern about the *swapping problem*, but about a limited quantity of case memory. When the memory is full, it becomes a barrier for the case-based system. Even when a limited number of stored cases are defined, the problem can still appear. It happens because there are cases with different sizes and they, even limited, can use different quantities of memory.

One could argue that the problem does not exist because the quantity of memory can be augmented. However, this is true for single domains and for simulations in computers, but it is not completely realistic when case-based systems are implemented in micro machines or for complex domains.



(a) **(b)**

Fig. 1 - The refinement and forgetfulness (abstraction) processes. In both processes, each simple arrow indicates the flow, each gray box indicates the part in consideration and each a_i indicates an action.

4 The Refinement and Forgetfulness Processes

A method to control the used quantity of memory is necessary. Thus, a theory of case abstraction is introduced in order to delete some case details and to release memory for the new case storage. However, instead of being a method to improve efficiency for case retrieval process as in PARIS system [1], this theory is a method to control the size of the case memory. It is made by a definition of two processes: refinement and forgetfulness.
Both processes use a measurement of case utility, proposed by [6]:

$$Utility = (ApplicFreq * AverageSavings) - MatchCost\ . \qquad (1)$$

This metric, presented by formula 1, was used in [6] to delete cases with negative utility, through an analysis on the use of the case and its similarity costs. However, in this section we focus on the utility measurement in order to define the refinement and the forgetfulness processes. In addition, in section 5 we discuss about a deletion of a case and the related works.

The use of the transaction logic allows the definition of forgetfulness and refinement processes that are simple and domain independent. With an analysis of *Wi* and *Wf,* it is possible to determine the main action of a case:

Abstraction Levels
Levels down from original case level

```
Input: a case c to be stored
       a set M of cases
Return: cases to be abstracted

func forget_to_store(c,M): int
step = 0;
abscase=1;
Mb = size_case_memory;
while size(c) >available(Mb)
      & abscase ≠ null
  {step=step+1;
   abscase=abstract_cases(M);}
return(abscase).
```

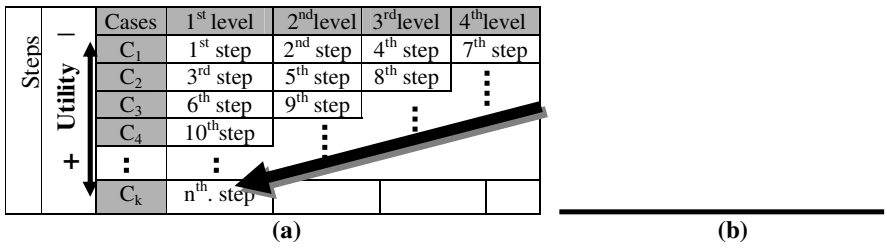


Fig. 2 - The steps of the ***abstract_cases*** function.

Definition 5: *The main actions of a case are those actions that are directly responsible by the deletions of those literal presented in Wi and directly responsible by the insertion of those literal in Wf.*

A general and simple example can illustrate the definition of the main actions of a case. Suppose an action called *act(X,Y)* that deletes the object *X* and insert the object *Y* in the world state. Consider the following plan that has *Wi = {a,b}* and *Wf = {c,d}*:

$$plan: act(a,g) \otimes act(g,h) \otimes act(h,j) \otimes act(j,c) \otimes act(b,d).$$

Thus, the case with the main action would be: $act(a,g) \otimes pln \otimes act(j,c) \otimes act(b,d)$.

A completely abstracted case is called remembrance-case (*Rc*). It has only the main actions of the plan and some *pln* instances filling the spaces between the actions. The main actions are obtained by the function called ***def_main_acts(pln,ϕ),*** where ϕ is the resulted plan by the planner after the adaptation of a case with pln instances in its structure. Otherwise, one completely refined is called detailed-case (*Dc*) and has all actions of the plan without *pln* instances in its structure.

The refinement process is given by the detailing of *pln* instances that are in a case. The fact that each *pln* instance represents a new sequence of actions as a sub-plan, it has its own main actions, *Wi* and *Wf*. The detailing is obtained, systematically, by the substitution of each *pln* instance by its respective main actions, as showed in figure 1a. When a case is refined it goes down one level in abstraction.

The refinement of a case is applied just when the case is retrieved and used to achieve a new goal. This process is repeated until the case becomes a detailed-case (*Dc*). This process changes the value of the utility of a case given by formula 1. When a case is refined, its utility increases because its *ApplicFreq* and *AverageSavings* increase and its *MatchCost* decreases. The algorithm is showed by figure 3a.

On the other hand, the forgetfulness process is responsible for the abstraction of a case as presented in figure 1b. A case will be abstracted when a memory becomes full and no more cases can be stored. The abstraction process has the obligation to release a quantity of memory in order to permit the storage of a new case.

```
Input: a case c to be stored; a plan p
   adapted from c; and a set M of cases

proc refinement(c,p,M)
for each pln in c
c' = subst(pln, def_main_acts(pln,p));
remove_from(c,M);
if size(c') > available(memory)
   then  M = forgefulness(c',M)
store(c',M).
```

```
Input:  a case c to be stored
        a set M of cases

func forgetfulness(c,M): set_cases
m = number_of_cases (M);
abscase = forget_to_store(c,M);
if abscase = null
      then M = case_deletion(c,M)
return(M).
```

(a) **(b)**

Fig. 3 - The refinement and forgetfulness algorithms.

The forgetfulness process has the ***abstract_cases*** function that, in the algorithm of figure 2b, executes the steps presented in figure 2a. A case can be abstracted just when the case with a lower utility immediately below has been abstracted twice more than it. This function returns *null* when no more cases can be abstracted.

The forgetfulness process, showed by figure 2a, does not consider *Rc* cases and performs the abstraction steps with the other stored cases. Figure 3b shows the forgetfulness algorithm.

However, there is a situation where no case can be abstracted, and it is impossible to release memory just by abstraction process. Therefore, when this situation happens, the forgetfulness algorithm must delete cases from the case memory. However, this deletion must be analyzed and controlled by a case-deletion policy.

4.1 The Coverage of Cases

To determine the coverage of a case, it is necessary to define the retrieval and adaptation phases. A case solves a problem just if it is selected and retrieved by the similarity metric and if it is adapted to solve all features of the new problem.

The most used rule to determine which cases are similar to a new problem or a case is the *Nearest Neighbor Method* that is based on a weighted sum of features. A typical algorithm for it can be found in [5]. A set of retrieved cases is defined as:

Definition 6: (Set of Retrievable Cases) *A set of all retrievable cases can be formed as the following:* $RetrieverSet(x,X) = N(x)$.
Where x is a case, X is a set of cases, $x \in X$, *and N(x) is a Nearest Neighbor formula.*

A suitable retriever for a CBP can be made by a similarity rule that just considers the features of *Wi* and *Wf*, respectively the initial and final states of a case. This consideration avoids that a case has different similarities for each level of abstraction.

The adaptation phase is important to make some necessary changes in a retrieved case in order to transform it into a solution for a new problem. Thus, the coverage of a case can be generally defined as:

Definition 7: (Coverage of a case) *The coverage of a case x in a set X of cases can be stated as:* $Coverage(x) = |\{x' \in X: x' \in AdaptationSet(x , RetrieverSet(x,X))\}|$.
In some case-based planning systems, the adaptation phase is made by a generative planning system that can create a plan from scratch, like MRL [4] and Prodigy/Analogy [10]. A generative planning as an adaptation phase can find a solution even if the retriever does not choose any case as the result of similarity.

Consequently, the deletion process just affects the coverage of the case-base, but it does not alter the system coverage.

If the adaptation phase is a generative planning system, it can adapt any case in X, and thus, $Coverage(x) = |N(x)|$. Generalizing the coverage for a set of cases:

$$Coverage(X) = |N(X)|. \qquad where:\ N(X) = \bigcup_{x \in X} N(x). \tag{2}$$

However, the deletion of high-coverage cases, even with an adaptation phase as a generative planning, can affect the system efficiency. Thus, the coverage of a case-base must to be preserved as high as possible.

With the definitions above, it is easy to notice that if the adaptation phase is a sound and complete generative planning and the retrieval phase is based on the initial and final states of cases, the processes of abstraction and refinement do not alter the coverage of the case base if they do not delete any case. However, if the refinement process deletes just a simple case, the total coverage can be decreased.

5 A Suitable Case-deletion Policy with a Lower Bound

A deletion-based algorithm is designed to choose cases to be deleted in order to keep the coverage closes to the optimal coverage. It can be made by a definition of a formula that, similar to ***benefit formula*** in [11], calculates the injury caused by a deletion:

$$Injury(x) = |N(x) - N(M\text{-}x) \cap N(x)| \tag{3}$$

The injury of a case is calculated by the analysis of the damage that can be caused in the total coverage $N(M)$ if x is removed from the case-base M.

In case-based planning, cases have different sizes and occupy different quantities of memory in their storage. However, it is possible to calculate the maximum number of cases that will be deleted and estimate the maximum loss of coverage.

Considering C_{max} the maximum size that a case can have in a certain domain, and C_{min} the minimum size, the number of cases with C_{min} that occupies the same space of a case with C_{max} is:

$$r = \left\lceil \frac{C_{\max}}{C_{\min}} \right\rceil. \tag{4}$$

In the worst case, a new case to be stored is a case with C_{max} size. Thus, the maximum number of cases that would be deleted by the looping *while*, in figure 4, is r.

```
func case_deletion (c, M): set_cases
   Determine N(x) for every case x ∈ M; set Mb = available(memory);
   While size(c) > Mb
        Select case c with minimal injury with respect to M (formula 3).
        M = M – c;
```

```
    Mb = available(memory);
return(M);
```

Fig. 4 - The case-deletion algorithm.

Considering that *m* is the total number of cases stored in a case memory. Thus, the maximum number of cases that will be deleted is a percentage of *m*: *D = r/m.*

However, we need to analyze how much coverage the algorithm decreases, and how much of it is more than the best (optimal) deletion.

In order to normalize notations, consider $\overline{X}$ as a set *X* of chosen cases to be deleted from the case-base *M*. In addition, consider the optimal choice as *B*, and the result presented by the algorithm as *A*. For example, there are, as optimal choices, the set *B* and the set $\overline{B}$, where $B \cup \overline{B} = M$. Where *B* is a set of remained cases by the deletion of $\overline{B}$ from *M*. The same with *A* and $\overline{A}$, resulted by the algorithm.

For a coverage of a set *X* of cases, the notation is $\boldsymbol{X^C}$ **= *Coverage(X).*** Intuitively, the lost coverage is the total coverage *N(M)* minus the coverage remaining. Thus, it can be defined as:

$$\overline{X}^{LC} = \textbf{\textit{LostCoverage}}(\overline{X}, M) = |N(M) - N(X)| . \quad (5)$$

With the definitions above, it is easy to prove that ***Injury***$(\overline{X})$ = *LostCoverage*$(\overline{X}, M)$. However due to space, the proof is omitted.

Figure 4 presents the case-deletion algorithm that requires $O(n^2)$ to be implemented, and it completes the abstraction and refinement algorithms. As Zhu and Yang´*s* algorithm, the case-deletion algorithm is a *greedy* algorithm and it does not perform the best choice at each step. However, the following theorem can be proved:

Theorem 8: *The optimal lost coverage is at least 58% of the lost coverage resulted by the case-deletion algorithm.*

Proof: *This proof is similar to [11], and due to space limitations, we omit some details. Suppose that r cases are deleted and that* $a1 \le a_2 \le ... \le a_r$ *are the injury of each case numbered by order of selection. Suppose that* $b_1 \le b_2 \le ... \le b_r$ *are the injury of each optimal case for deletion. Thus, the summation of the injuries results:*

$$\frac{\overline{B}^{LC}}{\overline{A}^{LC}} \ge \frac{1}{\left(\frac{r+1}{r}\right)^r - 1} \text{; With } r \to \infty, \ \frac{\overline{B}^{LC}}{\overline{A}^{LC}} \ge \frac{1}{e - 1} \to \frac{\overline{B}^{LC}}{\overline{A}^{LC}} \ge 0.58. \ \blacksquare$$

According to the theorem above, we have $\overline{B}^{LC} / \overline{A}^{LC} \ge 0.58$. However, after the deletion of *r* cases, *k* cases remain in the case-base. What is really important to reach a lower bound for coverage is the relation between the optimal coverage remaining and that one resulted by the algorithm, *i.e.*, the relation between B^C and A^C.

The definitions of coverage and lost coverage allow the following statement: $X^C + \overline{X}^{LC} = M^C$. The proof is simple and is obtained directly by the definitions. However,

Lecture Notes in Artificial Intelligence, 1952

when the cases are deleted, we can observe that B^C is a percentage of M^C, represented by z. It is possible to achieve a minimum value to z with respect to D:

Theorem 9: *If $r = D.m$ and $B^C = z.M^C$ then $z \geq (1 - 2.D)/(1-D)$ for $D < 50\%$..*

Proof: *Again, due to space limitations, we omit some proof details. Any set C with r cases in B has $C^{LC} \geq \overline{B}^{LC}$. The worst case is $C^{LC} = \overline{B}^{LC}$ .Thus, $\sum_{i=1}^{k/r} C_i^{LC} \leq B^C$ becomes $\frac{(1-2D)}{D}\overline{B}^{LC} \leq B^C$. Substituting $B^C = z.M^C$: $z \geq \frac{1-2D}{1-D}$.* ■

With the result reached by theorem 9, a relation between the optimal coverage and the number of deleted cases can be defined. Considering that $Y = A^C/B^C$; $X = \overline{B}^{LC}/\overline{A}^{LC}$ and $B^C = z.M^C$, the following formula is obtained:

$$Y = \frac{X(1-D) - D}{X.(1-2D)}. \quad (6)$$

To provide a lower bound better than that one found by Zhu and Yang [11], that is 0.63 for Y, is necessary to find the maximum value for D in the worst case. Thus, if $X = 0.58$ (worst case) and $Y \geq 0.63$, we can reach $D \leq 0.25$ by formula 6. With this result, if 1/4 of the stored cases is deleted, the lower bound is superior to 63%.

5.1 Relating Memory Size with Coverage Lower Bound

As analyzed previously, C_{max} is the maximum size that the deletion-based algorithm needs to release from memory. Considering that Mb is the maximum space in case-memory, and by formula 6 and with X=0.58, it is possible to write the following formula:

$$\left\lfloor \frac{Mb}{C\ \max} \right\rfloor \geq \left(\frac{1.724 - Y}{1 - Y} \right). r + 1 . \quad (7)$$

As C_{max} and C_{min} are known value, and r can be obtained from them by formula 4, the formula above becomes a relation between Mb and Y. Thus, it is possible to determine, for a desirable lower bound value, the size Mb of a case memory. Alternatively, for example, if some system has just a determined size Mb to store cases, it is possible to determine the lower bound for coverage depletion. For example, supposing $r = 500$, by formula 7, we will find that the memory must have space to store 1480 cases with C_{max} size in order to achieve a lower bound that is more than 63%.

6 Discussion and Conclusion

The theory presented in this paper makes possible the implementation of a case-based planning (CBP) with a limited size to store cases in memory. The majority of CBP systems consider an infinite case memory, as MRL [4], CAPLAN [7], and others. In some domains or for some applications as micro machines, the quantity of memory is limited and it can become a barrier for an effective case-based system.

Besides controlling the quantity of used memory, the theory presented in this paper, differently of the case deletion policy proposed by Smyth and Keane [8], handles with a lower bound for coverage depletion, using a suitable case-deletion policy. Zhu and Yang [11] reaches a lower bound of 63%, but they performed an addition-based algorithm that is not appropriate to handle memory size, because it is more efficient to delete few cases than choose a great number of them. Our approach has an advantage to establish a lower bound more than 63% for a fixed quantity of case memories, letting possible its application in any case-based system.

However, an analysis of the coverage resulted by the deletion process and the coverage resulted by the insertion of a new case will be investigated for future works.

References

1. Bergmann, R. and Wilke, W.: Building and refitting abstract planning cases by change of representation language. *Journal of Artificial Intelligence Research* 3 (1995) 53-118.
2. Bonner, A.J. and Kifer, M.: Transaction logic programming. *Technical Report, CSRI-323*, Department of Computer Science, University of Toronto (1995).
3. Francis, A.G. and Ram A.: The Utility Problem in Case-Based Reasoning. *Technical Report (ER-93-08)*. Georgia Institute of Technology, USA (1993).
4. Koehler, J.: Planning from Second Principles. *Artificial Intelligence,* 87. Elsevier Science. (1996).
5. Kolodner, J.L.: *Case-Based Reasoning*. Morgan Kaufmann. (1993).
6. Minton, S.: Qualitative Results Concerning the Utility of Explanation-based Learning. *Artificial Intelligence,* 42 (1990) 363-391.
7. Munõz-Avila, H. and Weberskirch, F.: Planning for Manufacturing Workpieces by Storing, Indexing and Replaying Planning Decisions. In: *Proceedings of AIPS-96.* AAAI Press. *(*1996)
8. Smyth, B. and Keane, M.: Remember to Forget: *A* Competence-preserving Case-deletion Policy for Case-based Reasoning Systems. In: *Proceedings of the International Joint Conference on Artificial Intelligence IJCAI'95*. (1995) 377-382.
9. Tonidandel, F. and Rillo, M.: Case-Based Planning in Transaction Logic Framework. In: *Proceedings of the Workshop on Intelligent Manufacturing Systems (IMS'98), 5TH IFAC*. Gramado, Brazil. Elsevier Science (1999).
10. Veloso, M.: *Learning by Analogical Reasoning in General Problem Solving*. PhD thesis, Carnegie Mellon University, Pittsburgh, USA (1992)
11. Zhu J. and Yang Q.: Remembering to Add: Competence-preserving Case-Addition Policies for Case-Base Maintenance. In: *Proceedings of the International Joint Conference on Artificial Intelligence IJCAI'99.* (1999)