Published in the Data & Knowledge Engineering 12 (1994) 277-296.

ACT-P: A Configurable Theorem-Prover

Ioannis Hatzilygeroudis
School of Engineering, Dept of Computer Engineering & Informatics, University of Patras, 26500 Patras, Greece.
e-mail: ihatz@grpatvx1.bitnet

Han Reichgelt
Dept of Computer Science, University of the West Indies, Mona, Kingston 7, Jamaica.
e-mail: han@uwimona.edu.jm

Abstract

There has been a considerable amount of research into the provision of explicit representation of control regimes for resolution-based theorem provers. However, most of the existing systems are either not adequate in that they do not allow the user to express any arbitrary control regime, or are too inefficient to be of practical use. In this paper a theorem prover, ACT-P, which is adequate but retains satisfactory efficiency is presented. It does so by providing a number of user-changeable heuristics which are called at specific points during the search for a proof. The set of user-changeable heuristics was determined on the basis of a classification of the heuristics used by existing resolution-based theorem provers.

Keywords: theorem prover, meta-level architecture, resolution control strategies, design methodology, adequacy.

1. Introduction

Some recent theorem proving systems, like e.g. OTTER [17], try to be more flexible than traditional systems as far as control of the proof search is concerned. They give the user the capability of participating in the determination of some of the components of the control regime to guide the search for a proof. However, they offer a hardwired flexibility which, in general, is not adequate. That is, one has a predefined number of hardwired choices in changing components of the control regime. Also, these systems do not allow to change the main resolution control strategy. However, such a choice may lead to a faster solution.

A type of a system architecture that allows for great flexibility in changing control knowledge and could be employed by a theorem prover is that used by meta-

© Copyright Elsevier Science B.V. All rights reserved.

level systems [1, 26]. Meta-level systems contain, in addition to an explicit representation of their domain knowledge, also an explicit representation of their control knowledge, knowledge about how to use domain knowledge to solve problems. Thus, any meta-level system consists of two levels, the object-level, at which the system reasons about the domain, and the meta-level, at which the system reasons about how to use the domain knowledge. Because one can explicitly reason about the way in which the object-level search space is explored, the hope is that one can improve the overall efficiency of the system by making more intelligent control decisions.

A distinction between meta-level architectures can be drawn based on the proportion of time spent at each level [26]. At the one end of the spectrum, we have object-level inference systems which spend most of their time at the object-level, e.g. [8]. Such systems contain a fixed object-level interpreter that looks for meta-level information at fixed points during its computational cycle. At the other end are pure meta-level inference systems. In such systems, computation is primarily done at the meta-level. In general, the meta-level contains a description of the object-level interpreter and it can use this description to simulate the behaviour of the object-level. An example is Socrates [6].

Pure meta-level systems provide the greatest flexibility. However, experience with Socrates clearly demonstrates the cost of this flexibility: unacceptable loss of computational efficiency. The reason for this is that the reduction in the object-level search space that can be achieved by explicit reasoning at the meta-level is more than offset by the time taken by the meta-level. This is called the "meta-level overhead problem". Thus, although the control decisions are more intelligent, the time taken to arrive at these decisions means that in most cases a less intelligent but hardwired control regime would be more efficient.

In this paper, a meta-level resolution-based theorem proving system, **ACT-P** (**A** **C**onfigurable **T**heorem-**P**rover), is described which aims at providing great flexibility without leading to an unacceptable loss of efficiency. The basic idea is to

make available a skeleton resolution-based theorem prover in which specific steps of its computational cycle, corresponding to various components of its control regime, are programmable by the user. The programmable steps are implemented as user-definable functions[1].

In order to decide what programmable steps to make available, a classification of the heuristics used in existing resolution-based theorem provers was made. The hope was that by studying existing theorem provers it would be possible to specify the important components of the control regime of a theorem prover.

The organisation of the paper is as follows. In section 2, our classification of existing resolution control heuristics is presented. In section 3, the design and implementation of ACT-P is discussed. Section 4 deals with the adequacy of ACT-P. Section 5 illustrates how ACT-P can be configured by providing some examples. Section 6 discusses related work, and finally section 7 concludes and speculates about possible extensions to ACT-P.

2. A Classification of Resolution Control Heuristics

We analysed a large number of control regimes used in resolution-based theorem provers and constructed a classification of the different types of heuristics employed. This was then used in designing ACT-P's computational cycle and in determining an adequate set of meta-functions (see next section). Figure 1 presents our classification. Each class of heuristics constitute a node of the classification tree. In the following, our classification is justified.

In order to make discussion in this and subsequent sections more comprehensible, some terminology is recalled. When the resolution rule is applied to two resolvable clauses φ and ψ to produce the 'resolvent' χ, then we refer to φ as the 'left parent' and ψ as the 'right parent' of χ. Also, φ and ψ together are referred to as a 'resolution pair'.

We distinguish four major classes of heuristics. The first major class, *resolution restricting strategies*, concern the generation of the search space and are used to

increase efficiency by restricting the size of the search space. Resolution restricting strategies comprise two subtypes, namely parent selection strategies and clause elimination strategies.

Resolution restricting strategies
- Parent selection (*)
  - Clause-based
  - Set-based
- Clause elimination (*)
  - Self-elimination
  - Interelimination

Resolution search strategies
- General search
  - Non-iterative (*)
  - Iterative (*)
- Resolution ordering (*)
  - Clause-based
  - Literal-based

Resolving strategies
- Resolving preparation (*)
  - Clause transformation
  - Literal ordering
- Resolving operation
  - Literal selection (*)
  - Resolvent construction (*)

Termination heuristics
- Failure heuristics
  - Infinite path detection (*)
  - Complexity checking (*)
    - Clause-based
    - Literal-based
  - Depth limit (*)
    - Temporary
    - Permanent
- Success heuristics
  - Node-based (*)
  - Process-based (*)

Fig. 1. A classification of resolution control heuristics[2].

Parent selection strategies are based on the observation that not all possible resolvents have to be constructed to be able to derive the empty clause. They therefore impose restrictions on the clauses to be selected for resolution. Parent selection strategies have also been called 'refinement strategies' [18] or 'restriction strategies' [27]. The criterion for selection can be either *clause-based* or *set-based*. In the former, the relevant criterion for selection applies to individual clauses. Thus, in P1-Resolution [22] one insists that resolution can only be applied if the left

[2] Denotation of the asterisks is explained in Section 3.2.

parent is a positive clause. In set-based parent selection strategies, on the other hand, one selects the parents from a set of known clauses. For example, in input resolution, the left parent must always be an axiom [4].

The second type of resolution restricting strategies, *clause elimination strategies*, aim to eliminate clauses that will not be useful in further search. They are also called 'simplification strategies' [18] or 'deletion strategies' [9]. We distinguish between *self-elimination strategies* which inspect clauses independently of the other clauses in the knowledge base (such as tautology elimination), and *inter-elimination strategies* which inspect the clause in conjunction with other clauses in the knowledge base. Examples of the latter are forward and backward subsumption [21].

The second main class of resolution control heuristics, *resolution search strategies*, concern the way the search space is searched. We distinguish between general search strategies and resolution ordering strategies. *General search strategies* traverse the search space in a blind way, without taking into account any resolution or domain or problem specific knowledge. We further divide general search strategies in *non-iterative* and *iterative* search strategies. The basis for this distinction is whether the strategies require only one generation of the search space (non-iterative) or more than one (iterative). The non-iterative strategies include the well-known breadth-first and depth-first strategies. An example of an iterative strategy is depth-first iterative deepening [13].

Unlike general search strategies, *resolution ordering strategies*, aim to increase the efficiency of the theorem proving process by judiciously ordering potential resolutions. Clearly, such ordering strategies presuppose some ordering criterion. Best-first type strategies belong to this class. We distinguish between two types of resolution ordering strategy, based on whether the criterion is *clause-based* or *literal-based*. An example of the former is unit preference; an example of the latter is weighting [27].

Whereas resolution search strategies concern the way in which the search space is searched, our third main class of heuristics, *resolving strategies*, concern the way in

which resolutions are performed. They are typically used in conjunction with a specific parent selection strategy. We draw a distinction between *resolving preparation* and *resolving operation* strategies. The former concern operations on clauses before the actual application of the resolution rule. We further distinguish between *clause transformation* and *literal ordering* heuristics. The first class are needed for resolution strategies that use more complicated representations than simple clauses, such as chains used for example in model elimination [15]. Literal ordering heuristics are used to order literals in parents prior to resolution. An example can be found in [20].

Resolving operation heuristics are used during actual resolution. We distinguish two classes, namely *literal selection* and *resolvent construction*. The former concern the number and/or type of literals involved in the resolution. Thus, Prolog will always try to find a pair of clauses that only resolve on their first literal. The latter concern the way in which parents are merged after resolution. Do we simply append the remaining literals of both parents, or do we eliminate redundant literals, as in merging [2]?

The final major class of heuristics, *termination heuristics*, concern the conditions under which search should be terminated. We distinguish between heuristics for detecting failure and heuristics for detecting success. *Failure heuristics* answer the question whether a node should be considered a dead end. If there is the situation where no further resolutions are possible, then the node is a dead end, but there are other reasons why a dead end has been reached. For example, a node is a dead end if it unifies with an earlier node on the current path.

Other failure heuristics prune a particular branch based on the complexity of the literals or clauses involved. For example, a *literal-based complexity checking heuristic* may impose a limit on the number of times the same function symbol may occur in a term. A *clause-based complexity checking heuristic* may prune a branch if the number of literals in a clause exceeds some limit.

A third class of failure heuristics restrict the depth to which a tree is searched. We distinguish between those cases where we have a *permanent depth limit* and those where we have a *temporary depth limit*. In the first case, we never perform resolution if it takes us over the limit. In the second case, we will in general first perform those resolutions that stay within the limit. Typically, depth limit heuristics are used in conjunction with a depth-first search strategy.

The second class of termination heuristics, *success heuristics*, come in two kinds. *Node-based* success heuristics are used to determine whether a node is successful. Usually, a node is successful if it contains the empty clause. However, one can imagine other possibilities. For example, in model elimination a node succeeds if the set of B-literals is empty, independent of the set of A-literals. *Process-based* success criteria are used to control whether search for solutions should be exhaustive or not.

3. The Design and Implementation of ACT-P

We have designed and implemented a skeleton theorem prover based on the ideas discussed in the previous sections, called ACT-P (A Configurable Theorem-Prover). ACT-P has been implemented in CommonLisp [24] on a Unix Sun workstation[3]. On the one hand, ACT-P is a resolution-based theorem prover in which the different components of its (overall) control regime are (re)definable by the user, via the (re)definition of a set of Lisp functions, called *meta-functions*.

On the other hand, ACT-P is a logic-based meta-level system based on a partial reflective architecture [26]. The system is reflective because it allows one to move from the object-level to the meta-level interpreter; it is partial because the overall computational cycle of the object-level interpreter is fixed and one can only ascend to the meta-level at specific steps in the computational cycle. Thus, the meta-level overhead is reduced [26]. ACT-P's object-level language is classical first-order predicate calculus (FOPC), whereas its meta-level language is CommonLisp enhanced with specially designed primitives, called *meta-primitives*.

3.1 Representation of the search space

The resolution search space in ACT-P is represented as an OR tree (see fig.2). The root (S) of the tree represents the initial state of the problem to be solved. For example, in fig.2 the axioms C_1, C_2, C_3 and the query (theorem) T_1, in clausal form, constitute the initial state. Each branch of the tree represents a potential resolution step and is labelled with the resolvent to be produced if the step is executed (e.g. R_{11}, R_{13}). Each node of the tree represents the set of clauses after the execution of the corresponding resolution step (e.g. S_{11}= S U $\{R_1\}$). The branches from a parent node to its child nodes represent the (new) potential resolutions due to the resolvent from which the parent node was produced (e.g. R_{11}, R_{12} are the potential resolvents from S_{11} due to resolutions of R_1 in S_{11}).

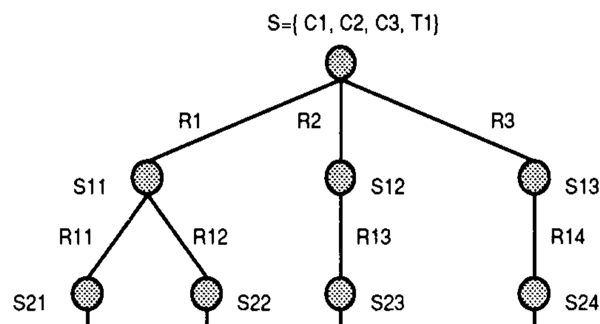

Fig. 2. Representation of resolution search space in ACT-P.

A path from the root to a leaf is a sequence of states (sets of clauses). In a solution path, the last sequent contains a solution clause (in most strategies the empty clause). In the actual implementation, we only keep a record of the produced resolvents, after the initial state, so that a solution path is a sequence of clauses (resolvents) in which the last sequent is a solution clause.

Also, information about a potential resolution step (branch) from a state (node) is stored in a step point. A *step point* is a 7-tuple, $< p_1, p_2, l_1, l_2, s_1, s_2, d >$, that

keeps information about a resolution step consisting of the left parent clause (p_1), the right parent clause (p_2), the literal(s) in the left parent that resolve (l_1), the literal(s) in the right parent that resolve (l_2), the substitution to be produced from the resolution of p_1 and p_2 (s_1), the substitutions produced up to this point (s_2), and the depth in the proof tree (d). We store pointers in a step point rather than the actual copies of the elements to improve space efficiency. The actual elements are retrieved from the knowledge base (see section 3.3) via an indexing mechanism.

3.2 The choice of the meta-functions

The classification presented in the previous section was the base for constructing the computational cycle of ACT-P. First, it guided us to incorporate all the necessary control components in it (see next subsection). Second, the decision of exactly which programmable steps (hence meta-functions) in the computational cycle should be made available to the user was based on it.

The first question that we faced in this decision was what level of abstraction in the classification we should make available for change. There are a number of trade-offs here. For example, if we were to choose classes of heuristics (nodes) that are lower on the classification tree, then corresponding steps in the computational cycle will be smaller and easier to program. Also, because the overall computational cycle will be more tightly structured, the system will be more efficient. On the other hand, there will be more steps to be specified, and this will increase the complexity of the system. Also, the system will be less flexible, since the programmable steps will be smaller and less abstract. Hence, there is a fundamental trade-off here between the number of the steps (functions) and their width (size), or in other words between efficiency and flexibility.

On the basis of the above considerations, we distilled the following principles to guide us in the choice of the appropriate programmable steps (meta-functions):

1. Avoid programmable steps corresponding to leaves in the classification tree.

2. If adding a step corresponding to a node results in inefficient code, then provide programmable steps corresponding to its children.
3. If adding steps corresponding to sibling nodes results in low flexibility, then provide a programmable step corresponding to their parent.

These principles led us to provide programmable steps for the classes of heuristics marked with an asterisk in fig.1.

Another question that we faced, regarding the above decision, concerned the number and the size of the steps (functions) needed to represent each of the selected nodes. There are a number of points here. First, not every node can be represented by just one meta-function since the corresponding heuristics may act in more than one point in the computational cycle. Parent selection strategies provide such an example (see Appendix, table 1.1). On the other hand, some of the heuristics, although they correspond to different nodes, are of the same nature and thus can be unified in the same programmable step. This is the case e.g. for resolution ordering, depth limit and non-iterative search strategies, which are represented by the same meta-function (see Appendix, table 1.1).

Also, because ACT-P's proof procedure consists of two stages (see next subsection), ACT-P provides two different meta-functions for several of the classes of heuristics, one of which is called in the first stage, the other in the second stage. Additionally, there are strategies that suggest use of different heuristics for left and right parents. ACT-P therefore provides two different meta-functions for heuristics that operates on the parents of a resolvent. Furthermore, one of the meta-functions has been reduced to just a global variable, called a *meta-variable*, because almost all of its action is hardwired. This is the case for the iterative search strategies.

Finally, apart from binary resolution [21], ACT-P offers two other inference rules, namely factoring and multiliteral resolution, mainly to assure completeness of the resolution refutation. Thus, two meta-functions and one meta-variable are provided for this purpose (see Appendix, table 1.2). For a brief description of all ACT-P's meta-functions and meta-variables see the Appendix.

In addition to the meta-functions, ACT-P also provides a number of auxiliary functions, the meta-primitives, which are intended to facilitate writing meta-functions. Thus, there are meta-primitives for testing a literal or performing simple operations on a literal. Likewise, there are a number of meta-primitives for testing properties of or performing operations on clauses. Finally, there are meta-primitives for getting at the various elements in a step point. For the use of a few of the meta-primitives see in the examples section. For a fuller description of the meta-functions and the meta-primitives, the reader is referred to [12].

3.3 The proof procedure

Logical formulas are introduced in the system via the primitive function 'store-assertions', which takes as arguments any number of FOPC formulas in Cambridge Polish notation with as connectives {~ , & , V , =>} and quantifiers {forall , exists}. The formulas are automatically transformed into clausal form and stored in the *knowledge base* as *axioms*. The knowledge base is a set (S_{kb}) containing, at any time in a proof process, all the clauses (axioms and produced resolvents) involved in the process up to that time. In order to prove some goal, the primitive function 'act-prove' is used, which takes as argument the *query* (*theorem*) to be answered (proved).

The active clauses from the knowledge base (i.e. those not excluded from the process for some reason), at any time, are distributed between two (possibly overlapping) sets, the *left parents* (S_{lp}) and the *right parents* (S_{rp}). These sets contain the potential left and right parent clauses. Their actual elements are pointers to the knowledge base rather than the clauses themselves. Each new resolvent is tried to be resolved with the clauses in one of the two sets, depending on the parent selection strategy followed.

The *agenda* is an ordered set of step points, that is a set containing, at any time, all potential resolutions in an order. Order is specified by the meta-function

'combine-points'. The *solution stack* is a set containing, at any time, the solutions already found. A *solution* is a set of variable bindings.

We distinguish two stages in the proof procedure, the *preparation stage* and the *main proof cycle*. This distinction is necessary because some resolution-based control regimes act in two stages. For example, in linear input resolution, we initially want to produce all possible resolutions. However, subsequently we only produce resolutions that have as their left parent the most recent resolvent, and as their right parent one of the initial axioms.

The computational cycle of ACT-P can be described by the following steps.

Preparation stage

1. Filter S_{kb}
2. Select (initial) S_{lp} and S_{rp}
3. Prepare S_{lp} and S_{rp}
4. Find resolution pairs between S_{lp} and S_{rp}
5. Make step points and put them on the agenda
6. Order the agenda
7. Store the initial state and (re)select S_{lp} and S_{rp}

Main proof cycle

8. If the termination condition is satisfied, return the contents of the solution stack
9. Produce the resolvent specified by the first step point on the agenda whose neither parent has been eliminated, then prune the point
10. If the resolvent is a solution clause, store corresponding solution on the solution stack and go to step 8
11. If iterative search is used and current situation fulfils the iteration condition, reset agenda, S_{lp}, S_{rp} and S_{kb} to the initial state, then go to step 8
12. If the resolvent is to be eliminated or does not pass the complexity test(s), go to step 8
13. If the resolvent leads to an infinite path, go to step 8
14. Find the new resolution pairs due to the produced resolvent
15. Make the new step points and add them to the agenda
16. (Re)order agenda
17. Update S_{kb}, S_{lp}, S_{rp}, with the produced resolvent
18. go to step 8

Which meta-function(s) is(are) involved in which step(s) is illustrated in the Appendix.

4. Adequacy of ACT-P

The question that dogs any partial reflective architecture is how one can be sure that the programmable steps are adequate in the sense that a sufficient number of different control regimes can be expressed by reconfiguring the basic skeleton interpreter.

One way of ensuring the adequacy of a system of this kind is by providing an abstract analysis of meta-level systems, as van Harmelen [26] does. He argues that in any logic-based meta-level system, one needs three sets of heuristics as far as the search strategy is concerned: a) a set of "generative" heuristics, which determine which part of the theoretically possible search space will actually be generated, b) a set of "directional" heuristics, for determining how the generated space should be traversed, and c) a set of "termination" heuristics, for determining under what conditions the search along a branch can be terminated with either success or failure. Since van Harmelen's partial reflective interpreter provides hooks for each of those sets of heuristics, he thus has an abstract argument for the adequacy of his system.

We adopted a more empirical approach. Rather than provide an abstract analysis of resolution-based theorem provers, we have analysed a large number of such systems to determine classes (types) of heuristics. Since resolution is the oldest and probably still most widely used theorem proving technique, a significant portion of the space of possible control regimes for such theorem provers has been investigated. A classification of the types of heuristics that have been employed therefore provides a clear and adequate indication of the programmable steps that need to be incorporated in the computational cycle.

Comparing our analysis to that of van Harmelen's, we notice the following correspondences. Our resolution restricting and resolving strategies are roughly

equivalent to his generative heuristics. What we call resolution search strategies largely correspond to his directional heuristics. Finally, our termination heuristics are almost identical to his. However, our analysis is more fine-grained and thus results in a system of finer granularity as far as configurability is concerned.

We can summarise our arguments for adequacy of ACT-P as follows:

1. We have identified all the necessary control components of a resolution-based theorem prover via our systematic classification of the types of a large number of resolution control heuristics.
2. All those components are present in the computational cycle of ACT-P as programmable steps, so that the user has access to almost all information about the control regime used in the proof procedure.
3. All those components are fully programmable; the full power of CommonLisp is available for that purpose.

5. Example Configurations

In order to give a flavour of the way in which ACT-P can be configured, in this section we discuss how certain resolution control regimes can be realised in ACT-P. ACT-P contains simple default definitions for the meta-functions, which are called if the user does not redefine (override) them.

5.1 The basic Prolog prover

Prolog uses SLD-resolution as its (overall) control regime. In SLD-resolution the left parent in a resolution pair is always the most recently produced resolvent and the right parent is an axiom. Prolog starts off with the goal (theorem) to be proved as the initial left parent.

In the preparation stage, initially the sets of the left and right parents are determined:

```
(defun select-init-l-parents (theorems axioms)
     theorems)

(defun select-init-r-parents (theorems axioms)
     axioms)
```

Then, all possible resolutions are produced. Prolog uses binary resolution. Thus, the meta-functions used to determine whether we use any other rule are left unchanged (default behaviour).

Prolog always only try to resolve the first literal in a left parent with the positive literal of the axiom (a Horn clause). This is the literal selection strategy of Prolog. We implement it in ACT-P as follows:

```
(defun select-l-literals (l-parent)
     (list (car l-parent)))

(defun select-r-literals (r-parent)
     (remove-if-not #'positive-literal r-parent))
```

where 'positive-literal' is an ACT-P meta-primitive.

The resolution pairs produced in this way are used to produce corresponding step points that are put on the agenda. Just before entering the main proof cycle, the left and right parents are reset. Because left parents always have to be resolvents , and there are no resolvents produced during the preparation stage, the left parents are set to 'nil', while the right parents are left unchanged:

```
(defun select-l-parents (init-l-parents  theorems axioms)
     nil)

(defun select-r-parents (init-r-parents theorems axioms)
     init-r-parents)
```

We now enter the main proof cycle. The way in which Prolog constructs resolvents is straightforward. We simply append the remaining literals in the axioms to the front of the remaining literals in the left parent:

```
(defun construct-resolvent (l-parent r-parent l-lits r-lits binds)
     (apply-substs
               (append (remove-literals r-parent r-lits)
                         (remove-literals l-parent l-lits))
               binds))
```

where 'apply-substs' and 'remove-literals' are meta-primitives.

Prolog uses the default check for determining whether a node is successful, namely that the produced resolvent be the empty clause. We therefore do not need to

change the meta-function 'solution-node'. Similarly, the default behaviour is sufficient for determining whether the process is to be terminated: a solution is found or there are no further resolutions to be produced. Thus, the default definition of the meta-function 'termination-condition' does not have to be changed. For completeness sake, we repeat those definitions here:

```
(defun solution-node (resolvent)
     (null resolvent))

(defun termination-condition (back-points solutions theorems)
     (or (= (length solutions) 1)
          (null back-points)))
```

The new produced resolvent is used for generating further resolution pairs. However, we only want to resolve the new resolvent against axioms, i.e. right parents. We redefine the following functions:

```
(defun resolve-with-l-parents (resolvent step-point)
     nil)

(defun resolve-with-r-parents (resolvent step-point)
     t)
```

The step points produced in this way need to be added to the agenda. Prolog uses a depth-first search with backtracking on failure. In order to produce this behaviour, we insist that the (new) step points are always added to the front of the agenda:

```
(defun combine-points (new-points old-points)
     (append new-points old-points))
```

Backtracking is performed automatically by ACT-P.

Finally, the left and right parents never need to be updated. Thus, we simply have the following definitions:

```
(defun update-l-parents (resolvent l-parents step-point)
     l-parents)

(defun update-r-parents (resolvent r-parents step-point)
     r-parents)
```

5.2 Adding more control

The above configuration of ACT-P gives the implementation of the basic Prolog system. However, one can add other control facilities, for example like those used in IC-Prolog [5] or in MRS [11]. IC-Prolog uses control annotations to specify the order of the literals in the body of a Horn-clause depending on the query. The same is done in MRS via the use of control clauses.

For example, the ACT-P axiom (where symbols starting with a '?' are variables)

```
(=> (& (parent ?x ?y) (parent ?y ?z))
    (grandparent ?x ?z))
```

is represented in IC-Prolog as

```
[ (=> (& (parent x y) (parent y z))
      (grandparent x? z))
  (=> (& (parent y z) (parent x y))
      (grandparent x^ z?)) ]
```

where the symbols '?' and '^' after a variable are used as control annotations. In this case, if x is bound to a constant, the first form of the axiom is used, whereas if z is bound to a constant, the second.

In MRS, this is achieved via the following control clauses

```
(=> (Better p1 p2)
    (Before (R p1 q1 r1) (R p2 q2 r2)))

(=> (& (Const u) (Var v) (Var x) (Var y))
    (Better (Parent  u v) (Parent x y)))

(=> (& (Var u) (Const v) (Var x) (Var y))
    (Better (Parent u v) (Parent x y)))
```

where 'Before', 'Better', 'Var', 'Const' are built-in primitives, and 'R' represents the action of resolving two clauses p and q to produce a new clause r.

In ACT-P this can also be relatively easily and more generally implemented by redefining the 'construct-resolvent' meta-function as

```
(defun construct-resolvent (l-par r-par l-lits r-lits binds)
    (append (re-order-literals
                  (apply-substs (remove-literals r-par r-lits)  binds))
               (apply-substs (remove-literals l-par l-lits)  binds)))

(defun re-order-literals (r-par)
    (if (null r-par) nil
```

```
        (let* ((init-lit (car r-par))
                (s-lit (select-best init-lit (cdr r-par)))
                (rest-par (remove-literals r-par (list s-lit))))
             (cons s-lit (re-order-literals rest-par)))))

(defun select-best (init-lit rest-par)
     (if (null rest-par) init-lit
          (let ((next-lit (car rest-par)))
             (if (less-vars init-lit next-lit)
                 (select-best init-lit (cdr rest-par))
                 (select-best next-lit (cdr rest-par))))))

(defun less-vars (lit1 lit2)
     (let*  ((terms1 (get-arguments lit1))
             (terms2 (get-arguments lit2))
             (vars1 (length (remove-if-not #'var terms1)))
             (vars2 (length (remove-if-not #'var terms2))))
        (< vars1 vars2)))
```

where 'get-arguments' and 'var' are built-in primitives.

This implementation in ACT-P is more general than those in IC-Prolog and MRS, because in IC-Prolog and MRS one has to write as many control statements as the predicates which one wishes to control. In ACT-P, the above configuration is adequate for all similar cases. However, the disadvantage of ACT-P's approach is that the required code is less readable than the MRS and IC-Prolog code. This is due to the procedural nature of ACT-P's meta-language.

5.3 Subsumption

Subsumption is a vital, but computationally expensive, clause elimination strategy used in theorem proving. A clause C subsumes a clause D if there is a substitution θ such that $C\theta \subseteq D$. There are two forms of subsumption, forward subsumption and backward subsumption. In *forward subsumption* a newly derived clause (resolvent) is eliminated if it is subsumed by an already existing clause. In *backward subsumption* an already existing clause is eliminated if it is subsumed by a newly derived clause.

The meta-function for implementation of clause elimination strategies is 'check-resolvent'. To implement forward subsumption it is redefined as follows.

```
(defun check-resolvent (resolvent clauses)
     (for-subsumes resolvent clauses))
```

```
(defun for-subsumes (resolvent clauses)
   (dolist (clause clauses resolvent)
      (if (subsumes clause resolvent) (return nil))))
```

where the function 'subsumes' needs to be further defined. To this end, 'unify-literals' is a very useful meta-primitive, which takes as arguments two literals and returns the variable bindings (substitution), if the literals unify, and 'nil', otherwise.

To implement backward subsumption the 'check-resolvent' meta-function is redefined as follows.

```
(defun check-resolvent (resolvent clauses)
   (back-subsumes resolvent clauses))

(defun back-subsumes (resolvent clauses)
   (dolist (clause clauses resolvent)
      (if (subsumes resolvent clause)
         (remove-par-clause clause))))
```

where 'remove-par-clause' is a meta-primitive which removes 'clause' from the knowledge base.

If both forms of subsumption are to be used, then elimination by forward subsumption should be first tested. In this case, the above meta-function should be redefined as follows.

```
(defun check-resolvent (resolvent clauses)
   (back-subsumes (for-subsumes resolvent clauses) clauses))
```

6. Related Work

There are a number of logic-based systems that like ACT-P try to be more flexible. A first category includes resolution-based theorem provers like those developed by the Argonne group, e.g. LMA/ITP [16], AURA [23] and OTTER [17]. These are very different systems from ACT-P.

The OTTER system, for example, provides a number of hardwired inference rules (binary resolution, factoring, hyperresolution, UR-resolution and binary paramodulation), as well as a number of other hardwired search control choices, but has a fixed parent selection strategy (a kind of set of support strategy). In contrast,

ACT-P provides a limited number of hardwired inference rules (binary resolution, factoring, multiliteral resolution), but offers a great variety of user-defined resolution control strategies. In OTTER users have to choose among a predefined number of control choices, whereas in ACT-P they can themselves define a great variety of controls. So, OTTER offers less flexibility, but because more is hardwired, greater efficiency than ACT-P.

A second category includes logic-based systems that can be characterised as meta-level systems. At the one end of the spectrum we find systems like Socrates [6], a highly flexible system. The user can change not only the inference rules and the control regime, but the representation language as well. However, the cost for this high flexibility is an unacceptable loss in efficiency due to the enormous meta-level overhead [6].

At the other end, there are systems that provide the user with a number of specific predicates which can be used to change the behaviour of the theorem prover. Examples of such systems include Gallaire and Lasserre's Prolog-based system [8] and MRS [10]. They typically require writing a meta-program alongside the object-program and offer low flexibility. For example, Gallaire and Lasserre's system uses a fixed resolution search strategy (that of Prolog), and is thus restricted to Horn-clauses, but allows the user to change the way in which literals and clauses are ordered prior to resolution, or the way in which dead ends are detected.

In between the above ends, there are systems that are more related to ACT-P. MOL [7] provides programmable steps implemented as redefinable predicates and is an extension of Gallaire and Lasserre's system by additionally allowing control of the backtracking strategy. van Harmelen's partial reflection system [26] (PRS hereafter) is an extension to MOL in two respects. First, PRS allows the user to change the termination criteria. Second, PRS is no longer restricted to Horn-clauses, but has in fact been parameterised with respect to the object-level language. Finally, FRAPPS [25] is a resolution-based system that allows the user to specify a wide range of control strategies via redefinable Lisp functions.

However, ACT-P differs in a number of respects from these systems. A first difference concerns the generality of the systems. MOL is restricted to Horn-clause logic. ACT-P does not suffer from this restriction. Moreover, ACT-P allows one to change the basic resolution strategy (the parent selection strategy), whereas MOL is restricted to Prolog-type binary linear input resolution. However, ACT-P is less general than PRS, as PRS is parameterised with respect to the object-level language, and can therefore be used for other logics than FOPC and for other proof theories than resolution refutation. Finally, FRAPPS and ACT-P offer comparable levels of generality.

A common feature of MOL, PRS and FRAPPS is that all use an explicit representation of the object-level proof tree. Thus, they have to store explicitly the open goals for every node. ACT-P uses an implicit representation. This allows for a more efficient, but less flexible, implementation of the proof procedure in ACT-P.

The set of meta-functions of ACT-P is very different from the set of redefinable predicates of PRS. As far as directional and termination heuristics are concerned (see section 4), PRS' meta-predicates are broadly comparable to ACT-P's meta-functions, although ACT-P's are slightly more specific. However, the most striking difference between PRS' predicates and ACT-P's functions concerns the generative heuristics. ACT-P offers an appreciably larger number of meta-functions, because it is based on a more specific analysis than PRS. This means that ACT-P has a larger number of programmable steps, which in general correspond to smaller step widths. Hence, ACT-P has a more strictly structured computational cycle. This has two consequences. First, it is much more complex to implement resolution-based systems in PRS. Second, ACT-P is more efficient for resolution-based systems.

FRAPPS and ACT-P are very similar systems in that both provide a number of user definable Lisp functions to specify a variety of resolution control strategies. However, in FRAPPS, as in PRS, the user constructs its own proof procedure using the offered functions, whereas in ACT-P there is a fixed skeleton of the proof procedure and there are certain steps that can be specified by the user. Although this

makes FRAPPS more flexible, it may lead to inefficiency. Also, because FRAPPS is not based on a systematic analysis, there is no argument given about its adequacy. So, it seems not to provide any functions for implementing resolving strategies and success heuristics. FRAPPS uses the notions of 'priority queue' and 'derivation graph'. The former is similar to ACT-P's 'agenda'. However, ACT-P uses the 'left parents' and 'right parents' instead of 'derivation graph'. This, although lacks some of the flexibility of directly manipulating the proof tree, results in higher efficiency, because the produced resolvent is tried for new resolution pairs with either the left or the right parents, and not with all parents as in FRAPPS.

7. Conclusions

In this paper, ACT-P, a configurable theorem prover, which attempts to combine the flexibility afforded by meta-level systems with acceptable efficiency[4], has been discussed. ACT-P is an example of a partial reflection architecture. It makes available a number of user-definable functions that are called at specific points during its computational cycle. As a result, the meta-level overhead is kept within acceptable bounds. Also, given that the meta-language of ACT-P is the same as its implementation language, redefinition of the meta-functions overrides their default definitions without any significant loss in efficiency, since no extra interpretation is required.

To assure adequacy of ACT-P, we analysed a large number of control regimes used in resolution-based theorem provers and classified the types of the various heuristics employed in a way that guided us in deciding which steps should be made programmable. The fact that ACT-P's meta-functions are closely related to that classification suggests that ACT-P is indeed adequate.

ACT-P offers great flexibility in incorporating heuristics involved in resolution-based proofs. Various well-known general search strategies such as breadth-first, depth-first, best-first and their variants, various resolving techniques, and the majority of existing resolution search strategies as well as their combinations can be

implemented in ACT-P. Also, because of its flexibility, ACT-P allows the user to define a great variety of problem specific heuristics to be taken into account during the proof procedure.

Thus, ACT-P can be used as the basis of a tool for testing various strategies under a uniform implementation environment, to see their behaviour and suitability in different kinds of problems. In this respect, a library of different meta-functions definitions implementing various heuristics would be useful. Furthermore, an extension to ACT-P that would automatically choose the most promising heuristic, would be very interesting.

A weak point of ACT-P's present implementation is due to the fact that in ACT-P all potential resolution pairs from a node in the search space are first found and stored (as step points). It is only after this that the actual resolution takes place to see if any of them leads to a solution. Thus, although an early pair may lead to a solution, all the other pairs have to be found and stored, whereas in a hardwired approach the first resolution pair found is resolved, then the next, and so on. If the current pair leads to a solution, the remaining potential pairs need not to be found. This is actually the price paid for the great flexibility.

Also, ACT-P provides only a limited number of inference rules. Thus, another extension could be the provision of multi-clause resolution rules, such as hyperresolution and Unit-Resulting resolution, as well as rules concerning equality, such as demodulation and paramodulation [27].

Finally, another weak point of ACT-P that needs further work is due to the fact that ACT-P offers no direct constructs for attaching information to literals and terms. Thus, ACT-P can incorporate strategies like Lock resolution [3] and Connection graphs [14] only with great difficulty, if at all.

Acknowledgements

We would like to express our thanks to Frank van Harmelen for extensive comments on an earlier stage of this work. Most of this work was done when both authors were

at University of Nottingham, UK. The first author was financially supported by the State Scholarships Foundation (SSF) of Greece.

Appendix: Meta-functionsand meta-variables of ACT-P

Table 1
(Corresponding to resolution control heuristics)

class of heuristics	function/variable name	inputs/values	output	step involved
parent selection	select-init-l-parents	query, (initial) axioms	(initial) left parents	2
	select-init-r-parents	"	(initial) right parents	2
	select-l-parents	(initial) left parents, (initial) axioms, query	left parents	7
	select-r-parents	(initial) right parents, (initial) axioms, query	right parents	7
	resolve-with-r-parents resolve-with-l-parents	resolvent, step point	't' or 'nil'	14
	update-l-parents	resolvent, step point, left parents	left parents (updated)	17
	update-r-parents	resolvent, step point, right parents	right parents (updated)	17
clause elimination	eliminate-axioms	(initial) axioms,query	axioms (filtered)	1
resolving preparation	prepare-l-parents	left parents	left parents (prepared)	3
	prepare-r-parents	right parents	right parents (prepared)	3
literal selection	select-l-literals	left-parent	selected literals	4 & 14
	select-r-literals	right parent	selected literals	4 & 14
resolvent construction	construct-resolvent	left parent, right parent, resolvable literals variable bindings	resolvent	9
non-iterative search, resolution ordering, depth limit	combine-points	new step points, agenda	agenda (updated)	6 & 16
infinite path detection	detect-infinite-path	resolvent, current path resolvents	't' or 'nil'	13
complexity checking, clause elimination	check-resolvent	resolvent, all parents	resolvent or 'nil'	12
node-based success	solution-node	resolvent	't' or 'nil'	10
process-based success	termination-condition	agenda, solution stack, query	't' or 'nil'	8
iterative search	*iteration-step*	positive integer or 'nil'	(meta-variable)	11

Table 2
(Corresponding to inference rules)

inference rule	function/variable name	inputs/values	output	step involved
factoring	factor-l-parent	left parent	't' or 'nil'	4 & 14
	factor-r-parent	right parent	't' or 'nil'	4 & 14
multiliteral resolution	*multi-literal-resolution*	't' or 'nil'	(meta-variable)	4 & 14

References

[1] L. Aiello, C. Cecchi and D. Sartini, Representation and Use of Metaknowledge, *Proceedings of the IEEE* 74 (1986) 1304-1321.

[2] P. Andrews, Resolution with merging, *JACM* 15 (1968) 367-381.

[3] R. S. Boyer, Locking: a restriction of resolution, PhD Thesis, University of Texas at Austin, Austin, TX, 1971.

[4] C.-L. Chang, The unit proof and the input proof in theorem proving, *JACM*, 17 (1970) 698-707.

[5] K. L. Clark and F.G. McCabe, The control facilities of IC-Prolog, in: D. Michie, ed., *Expert systems in the micro electronic age* (Edinburgh University Press, Edinburgh, UK, 1979) 122-149.

[6] R. Corlett, N. Davies, R. Khan, H. Reichgelt and F. van Harmelen, The architecture of Socrates, in: P. Jackson, H. Reichgelt and F. van Harmelen, eds, *Logic-based knowledge representation* (MIT Press, Cambridge, MA,1989).

[7] K. Eshghi, Meta-language in logic programming, PhD Thesis, Imperial College of Science and Technology, London, UK, 1986.

[8] M. Gallaire and C. Lasserre, Meta-level control for logic programs, in: K. Clark and S. Tarnlund, eds, *Logic Programming* (Academic Press, New York, 1982).

[9] M. Genesereth and N. Nilsson, *Logical foundations of AI* (Morgan Kaufmann, Los Altos, CA, 1987).

[10] M. R. Genesereth, MRS - A meta-level representation system, Technical Report HPP-83-27, Heuristic Programming Project, Stanford University, Stanford, CA, 1983.

[11] M. R. Genesereth and M.L. Ginsberg, Logic Programming, *CACM* 28 (1985) 933-941.

[12] I. Hatzilygeroudis, Integrating Logic and Object for Knowledge Representation and Reasoning, PhD Thesis, University of Nottingham, UK, 1992.

[13] R. Korf, Depth-first iterative deepening: An optimal admissible tree search, *AI* 27 (1985) 97-109.

[14] R. Kowalski, A proof procedure using connection graphs, *JACM* 22 (1975) 572-595.

[15] D. Loveland, A simplified format for the model elimination procedure, *JACM* 16 (1969) 349-363.

[16] E. Lusk and R. Overbeek, The automated reasoning system ITP, Technical Report ANL-84/27, Argonne National Laboratory, Argonne, Illinois, 1984.

[17] W. W. McCune, OTTER 2.0 User's Guide, Technical Report ANL-90/9, Argonne National Laboratory, Argonne, Illinois, 1990.

[18] N. Nilsson, *Problem solving methods in Artificial Intelligence* (McGraw-Hill, New York, 1971).

[19] H. Reichgelt and N. Shadbolt, A specification tool for planning systems, *Proceedings of the 9th ECAI*, 1990, 541-546.

[20] R. Reiter, Two results on ordering for resolution with merging and linear format, *JACM* 18 (1971) 630-646.

[21] J. Robinson, A machine-oriented logic based on the resolution principle, *JACM* 12 (1965) 23-41.

[22] J. Robinson, *Logic: Form and Function* (Edinburgh University Press, Edinburgh, UK, 1979).

[23] B. Smith, Reference manual for the enviromental theorem prover: An incarnation of AURA, Technical Report ANL-82/2, Argonne National Laboratory, Argonne, Illinois,1988.

[24] Guy L. Steele JR., *Common LISP: The Language* (Digital Press, 1984) .

[25] T.E. Uribe, A.M. Frisch and M.K. Mitchell, An Overview of FRAPPS 2.0: A Framework for Resolution-based Automated Proof Procedure Systems, *Proceedings of the 11th International Conference on Automated Deduction (CADE-11)*, 1992.

[26] F. van Harmelen, *Meta-level Inference Systems* (Pitman, 1991).

[27] L. Wos, R. Overbeek, E. Lusk and J. Boyle, *Automated Reasoning: Introduction and Applications* (Prentice-Hall, Englewood Cliffs, NJ, 1984).