Or-Parallel Execution Models of Prolog

David H.D. Warren

Department of Computer Science
University of Manchester, Manchester M13 9PL

1 Introduction

Multiprocessor machines based on state-of-the-art microprocessors hold promise for delivering much higher computing power at a reasonable cost. However, such machines will not gain wide acceptance unless they can be viewed essentially as "black boxes" that simply run software faster. There are enough complexities in software development without burdening the programmer with the further problem of explicitly managing parallelism. The challenge, therefore, is to exploit parallelism in application programs in a way that is *invisible* to the programmer, and that does not incur major overheads.

In general, this is an extraordinarily difficult problem, since most programming languages are based on von Neumann constructs that are inherently sequential. It is therefore natural to turn to non von Neumann languages. In this context, Prolog stands out as being uniquely qualified to unlock the potential of multiprocessors. It is one of the few, if not the only, non von Neumann language that is already in wide use for serious applications. Indeed, its popularity is based predominantly on its suitability for programming advanced applications. The basic language (Horn clauses, annotated with goal ordering plus cut) does not include any von Neumann constructs, yet is sufficiently powerful that it represents the bulk of applications code.

Thus, the general goal, towards which this paper is directed, is how to exploit parallelism transparently in Prolog. Prolog offers two main kinds of parallelism: or-parallelism and and-parallelism. In this paper, we focus on or-parallelism. We will examine different models for executing Prolog in or-parallel on shared memory multiprocessors. We focus on or-parallelism in particular, since it seems to offer good potential for large-scale, large-granularity parallelism across a wide range of applications. Or-parallelism typically manifests itself in the form of "generate and test" or "iterate and compute". Examples are querying a deductive database, parsing a natural language sentence, searching for a string in a document, or compiling a set of objects. Often, as in the last example, the or-parallelism is "wrapped up" in a 'setof' construct.

The main problem with implementing or-parallelism is how to represent different bindings of the same variable corresponding to different branches of the search space.

We will discuss a progression of possible implementations, starting from a simple abstract model, and moving to more complex models that aim to be more efficient. In this way, we intend to illuminate what the implementation choices are [6].

The models to be discussed are: (1) the Abstract Model of classical resolution theory which involves copying all inherited structure; (2) the Naïve Model, where bindings are stored in a simple chronological list; (3) the SRI Model, originally proposed by Warren at SRI, where each processor has its own binding array; (4) the Argonne Model, designed and implemented by Lusk and Overbeek at Argonne [4,1], where there is a hash array recording the bindings on each arc of the or-parallel tree; (5) the Manchester-Argonne Model, a variant of the Argonne model proposed by Warren, where hash arrays are only created as arcs become actually shared; (6) the Argonne-SRI Model, a variant of the SRI Model which uses the "favoured" binding approach of the Argonne Model.

Of particular concern, in comparing these models, are the costs of creating and accessing variable bindings. In standard Prolog implementations, these are both very fast operations that can be performed in constant time. Creating a binding is essentially just a write to memory (an assignment to the variable value cell), plus, in general, a "trailing" of the variable's address by pushing it onto a pushdown list called the "trail". Accessing a binding is nothing more than a memory read of the variable's value cell. It is vital to maintain comparable efficiency for these operations in an or-parallel implementation.

It is also vital to keep down the cost of creating multiple tasks at an or-parallel branch point. Ideally this overhead should be no greater than the cost, in a standard Prolog implementation, of backtracking through these alternative tasks. In a standard Prolog implementation such as the WAM [8,7,2], this creation of multiple tasks corresponds to the 'try', 'retry', 'trust' operations which create and restore choice points. All these are constant-time operations which can be implemented in, typically, 15 or 20 machine instructions. (There is also the cost of "untrailing" variable bindings, but this is probably better viewed as a (delayed) part of the cost of creating a variable binding).

In general, we want to keep the cost of all operations in a parallel implementation very close to what they would be in a sequential implementation. Otherwise, what we gain through parallelism can easily be lost in extra overheads.

2 The Abstract Model

A basic reference point in comparing or-parallel models is the classical "Abstract Model" of resolution theory (specialised to Horn clauses and Prolog control). We will describe it in some detail, mainly to "set the scene" for the other models.

The computation is viewed as a tree, where each node is labelled with a "task" consisting of a list of goals. For the root node, the task is just the initial query. Other nodes are labelled with derived tasks, the derivation process being resolution. Each node corresponds to one possible way of matching the leftmost goal of the parent node's task with a program clause. The derived task can be viewed as a copy of the parent task, with the leftmost goal replaced by (a copy of) the body of the matching clause, and with variable instantiations being made as required for the match.

Standard sequential Prolog explores the computation tree in a depth-first, left-to-right manner. At any point in time, the processor is "at" a particular node, working on the task at that node. The processor keeps a record of the branch of the tree above it. In an actual Prolog implementation, such as the WAM, the representation of the task tree is rather indirect. We will discuss the details a bit more later. However one can imagine a Prolog implementation which follows very directly the abstract model. The main difference between such an implementation and a "real" implementation is that it would keep an actual copy of the tasks on the branch above the current node, whereas the real implementation does not, but is able to reconstruct the earlier tasks as needed on backtracking.

Thus our abstract model has the (rather serious) overhead at each step of having to make a physical copy of the goals inherited from the parent task. However, if one accepts this, then it is rather easy to support or-parallelism.

Multiple processors can be simultaneously exploring different branches of the tree. One processor can start exploring the tree in the standard depth-first left-to-right manner. As soon as it creates a node with unexplored arcs remaining to the right, another processor can "steal" one of these arcs and start exploring the subtree beneath it, again in the standard manner. This process can continue until all the available processors are busy. When processors backtrack, they will of course avoid exploring arcs that have been "stolen" by other processors. When a processor backtracks to the point it started from, it becomes idle and available to explore any unexplored arc that currently exists. In general, we shall assume (in this and subsequent models) that it will never pick an arc if there is an ancestral node with unexplored arcs.

It is not strictly necessary for the processors to follow the standard Prolog search procedure. At any point they could relinquish the node they are currently at, and pick some other unexplored arc. However by following the standard search procedure as far as possible, each processor will for the most part act just like a sequential Prolog processor, and standard implementation techniques and optimisations can be applied. Ideally, the computation tree will be such that each processor will get large subtrees to explore, and only rarely will a processor have to "switch tasks" and jump to a different part of the tree. Thus the aim is to achieve very coarse-grained or-parallelism, where each chunk of work is very much like a standard Prolog

computation.

The disadvantage of the abstract model as a basis for a practical implementation is, as we have mentioned, the massive copying that would be necessary at each (parallel) branch point. This drawback would appear to rule it out as a practical approach. However, leaving this drawback aside, the model has some nice properties: the processors can work completely independently on physically separate data, and the implementation could in all other respects be the same as a sequential implementation.

3 The Naïve Model

There is a rather simple modification of the Abstract Model that is useful for understanding the later, more elaborate models. We will call this the "Naïve Model".

In this model, a task is represented as a list of goals together with a list of variable bindings. When we derive a new task by resolution, variable bindings are only applied to the new variables. Variables in the old goals are left unchanged. Instead corresponding bindings are added to the front of the bindings list. Thus the binding list records bindings in reverse order of their creation, i.e. most recent binding first, oldest binding last. Because there is no alteration of the old goals, the same physical copy can be shared between the parent and its offspring. Likewise, the offspring binding lists are simply extensions of the parent binding list. Thus the model takes good advantage of the kind of sharing of substructures that is familiar in Lisp and other symbolic languages.

There is a very close correspondence between the Naïve Model and a standard Prolog implementation. The binding list is analogous to the trail. Because a sequential Prolog implementation is only exploring one branch at a time, it can afford to instantiate the variables in the inherited goals, provided it "undoes" these instantiations before backtracking to an alternative branch. It uses the trail to perform this undoing. The trail can be thought of as a binding list where one records just the variable name, rather than a name/value pair, since the value is stored separately in the variable value cell. It would in fact add very little to the overall cost of making a binding to store the value as well as the address in the trail, although there would normally be no point in this.

A binding of a variable is said to be unconditional if the variable was created since the last parallel branch point. Such a binding will be the same for all processors having access to the variable. Bindings which are unconditional do not need to be recorded in the binding list. Instead, the binding is simply applied to the variable value cell. This is analogous to the way the trail is optimised in a standard Prolog implementation. A similar optimisation can be applied to all the models we shall discuss.

In the Naïve Model, one cannot look up a variable's value simply by referring to the value cell. Instead, one has to search through the binding list. The access time is therefore, on average, proportional to the number of bindings in the current binding list, (or, with a simple refinement, to the number of bindings that are more recent than the variable in question). Either way, the access time is not bounded. This is the Naïve Model's major drawback. To offset this it has some considerable advantages: it is conceptually simple, there is no overhead on setting up a parallel branch point, and processors can switch tasks easily without any task initialisation overheads.

4 The SRI Model

The "SRI Model", was first proposed by D.H.D. Warren at SRI in 1983, in private discussions and unpublished documents. It has much in common with a scheme proposed independently by D.S. Warren [9], and contains elements of the independently conceived "shallow binding" scheme of Miyazaki et al [3]. The SRI Model can be viewed as a modification of the Naïve Model, in which each processor has its own private binding array. The binding array can be considered to be a local memory attached to the individual processor. In this binding array, the processor "shadows" the bindings from the binding list it is currently working with. That is, the binding array contains essentially the same information as the binding list, stored in a form that gives constant-time access to the value of any variable. The binding array could be implemented simply as an ordinary array; the variables on a branch would be numbered in chronological order, and that number would be used as a simple index into the binding array. Alternatively, the binding array could be implemented as a hash table accessed by hashing on the variable address.

Bindings are added to the binding array at each resolution step, and are removed on backtracking. When a processor runs out of work and wants to switch tasks to some new node, it has to change the state of its binding array to reflect the state of the binding list at the new node. It could simply reinitialise its binding array from scratch. However, normally the new state will have a lot in common with the old state, and there is a more efficient way to achieve the required new state.

Conceptually, the processor backtracks to the common ancestral node, removing bindings from its binding array in the normal way. It then "skims" forward to the new node, installing the bindings that have already been found by the processor (or processors) that explored that path. Thus the bindings that correspond to the common portion of the old and new binding lists are left intact in the binding array. That is, any binding created before the common ancestral node is left untouched in the binding array. All that is happening is that bindings from the old binding list that are more recent than the common ancestral node are removed from the binding array, and bindings from the new binding list that are more recent than the

common ancestral node are copied into the binding array. In this way, the cost of switching nodes is proportional to the "distance" between them, or more precisely to the number of bindings on the path between them.

The advantage of the SRI model is that most operations are very similar to what they would be in a sequential implementation. In particular, variable access and binding are both constant-time operations, and do not cost much more than they would in, say, the WAM. [1]

The main disadvantage of the model is the cost of switching between tasks. In the worst case, a processor might switch back and forth between nodes on two long branches, duplicating work by continually removing and reinstating the same bindings. Thus the model is unsuitable for fine-grained parallelism, but should perform well if the processors can always pick large chunks of work to do.

5 The Argonne Model

The "Argonne Model", which Lusk and Overbeek have designed and actually implemented, was partly influenced by the SRI model. It can be viewed as taking an alternative, and very ingenious, approach to improving on the Naïve Model. The model has been implemented in C as an extension of the WAM, and the implementation has been designed to be easily portable across a range of shared memory multiprocessors: it currently runs on HEP, Sequent and Encore machines.

A key idea is the concept of a "favoured" binding. Each shared variable can be considered to "belong" to one of the processors, generally the one that created it. This processor is the "favoured" processor for that variable, and is allowed to actually bind the variable value cell. The cell is marked with a special bit to show that this binding is only relevant to the favoured processor.

The other, non-favoured, processors view the shared variable as being an "alien" variable, belonging to another processor. They have to look elsewhere than the value cell to find out whether the variable is bound as far as they are concerned. The mechanism used is a chain of hash tables. This can be viewed as an optimisation of the binding list of the Naïve Model. There is a hash table for each arc of the or-parallel tree, recording the bindings of shared variables made on that arc. The hash tables on a branch are kept in a chain. A binding can be looked up in this hash table chain in much the same way as in the Naïve Model's binding list, but faster, depending on the hash table size. As an optimisation, each processor keeps a private chain of those hash tables that are "relevant" to that processor. A hash

[1] In an idealised implementation with infinite processors where each processor is shadowed by another processor recording the same bindings, the execution time for a problem would be proportional to the length of the longest branch. In this rather unrealistic sense, the model can approach the best conceivable exploitation of or-parallelism.

table is relevant if it contains a binding of an "unfavoured" or "alien" variable, i.e. one not "belonging" to the processor concerned.

Variables that are not shared with other processors are said to be "private", and bindings can be implemented by unconditionally binding the variable value cell, as in the standard WAM. The variable may subsequently become shared, but all processors will see the same unconditional binding.

Corresponding to this classification of variables into "unfavoured", "favoured" and "private", the branch a processor is working on is divided into an "unfavoured" section, a "favoured" section, and a "private" section. The private section is the part beneath the last potential parallel branch point. The favoured section is the rest of the branch beneath the node at which the processor started when it last switched tasks, i.e. those arcs for which this processor is the "leftmost" processor working on that arc. The unfavoured section is everything above up to the root.

Let us now sum up what is entailed in the key operations. To access a variable: if the variable is private or favoured, we simply extract the value from the value cell, exactly as in the WAM; if the variable is unfavoured, we search for a binding in the chain of relevant hash tables (and we only need to search as far as the point where the variable was created, c.f. the Naïve Model). To bind a variable: if the variable is private, we simply assign to the value cell; if the variable is favoured, we assign to the value cell with a special mark, and enter the binding in the current hash table; if the variable is unfavoured, we just enter the binding in the hash table, and make the hash table relevant by entering it in the chain of relevant hash tables if this has not already been marked as done. To switch tasks is extremely simple: the processor just initialises its relevant hash table chain to be the general chain existing at the new node. (If the chains are implemented as linked lists, this is just a question of initialising a pointer).

The advantages of the Argonne Model are that there is a very low overhead on switching tasks, and there is quick, constant-time, access to most variables, except for those that are unfavoured. It is hoped that the proportion of variable references that are unfavoured will be small, provided the processors can pick relatively large tasks high up the tree. There is some evidence that this can be achieved in practice, but we will discuss this point further in the concluding section.

The main disadvantage of the Argonne Model is that the access time for an unfavoured variable is not bounded, but is proportional to the number of unfavoured bindings made so far on the branch. This is a rather worrying feature, since the number of such unfavoured bindings could grow indefinitely. The hash table mechanism reduces the cost of access, but does not cure the basic basic complexity problem. In the current Argonne implementation, the hash table size is fixed at 16, and the table is not expanded if it becomes "overfull"; instead, entries simply build up in one of 16 linked lists. This means that the hash table mechanism is only gaining a constant factor speedup of around 16 over a simple binding list as in the Naïve

Model. Even if hash tables were expanded dynamically, variable access would still be proportional to the number of relevant hash tables, and this number can also grow indefinitely.

A more minor disadvantage is that quite a lot of work is done building hash tables which are not really needed, since the arcs corresponding to these hash tables never become shared. However, this is of less concern, since it does not introduce any non constant-time overheads.

6 The Manchester-Argonne Model

A variant of the Argonne Model has been proposed by Warren. We will call this the "Manchester-Argonne Model". The aim of this model is to make access to unfavoured variables a bounded cost operation, while preserving the main features and advantages of the Argonne Model.

A key observation about the Argonne Model is that the hash tables are largely superfluous on arcs that are not actually shared. The Manchester-Argonne Model therefore does not create the hash table for an arc unless, and until, another processor wants to share that arc. The processor that creates the arc merely inserts the bindings in a binding list, as in the Naïve Model. Doing this is just a minor extension of the "trailing" operation in sequential Prolog. The second processor, which wants to share the arc, then creates the hash table it requires from the information in the binding list. It simply copies the bindings corresponding to the about-to-be-shared arc into a new hash table. At this point, the number of bindings is known, so the size of the hash table can be chosen to fit the number of entries it must contain, in contrast to the standard Argonne model. The cost of looking up a binding in a hash table can therefore be kept a constant-time operation, assuming a good hashing function. Each processor also needs a private binding array, similar to that of the SRI Model, to store the unfavoured bindings it itself makes.

When a node ceases to be a parallel branch point, because all but one of the subtrees beneath the node have been completely explored, the node can be discarded, and hash tables above and below the node merged. Bindings below are checked to see whether they are now private, in which case they can be applied to the variable value cell (and removed from the binding list); otherwise they are inserted in the hash table above. The size of the hash table above may be increased if necessary. Probably, hash table sizes should be a power of 2, so that the number of times an element is copied is kept logarithmic in the size of the array. The reason for doing this merging of hash tables is to keep bounded the number in existence at any time, and thereby to keep bounded the cost of variable lookup.

To look up an unfavoured binding, the processor first looks in its private binding array and then accesses the chain of hash tables corresponding to the shared arcs

above it. The cost is proportional to the number of shared arcs above it plus one. The number of such arcs cannot be greater than the number of processors [2], and is more likely to be of the order of the logarithm of the number of processors. Thus the cost of access is bounded (for a given machine) and likely to be relatively small.

The operations of accessing other variables and creating bindings are very close to what they would be in sequential Prolog. The operation of switching tasks may involve some overhead, but it is less than in the SRI Model. On switching tasks, a new hash array may have to be created, at a cost proportional to the number of bindings on the about to be shared arc. Note that this is work that the Argonne Model would have to do anyway at the time the bindings were originally created. One can think of the Manchester-Argonne Model as a "lazy creation of hash tables" version of the Argonne Model. Note also that there will be no significant overhead on switching tasks if the arc to which we are moving is already shared.

There is also the overhead on deleting a branch point of coalescing hash tables. This cost is generally proportional to the number of bindings on the arc below. There may also be the subsidiary cost of enlarging the hash table above, if necessary. In the worst case, this could make the cost of maintaining an unfavoured binding be logarithmic in the number of unfavoured bindings, but it seems unlikely the behaviour in practice would approach the worst case.

On balance, therefore, we feel the Manchester-Argonne Model is likely to perform significantly better than the Argonne Model.

7 The Argonne-SRI Model

It is possible to modify the SRI Model by introducing a treatment of favoured variables as in the Argonne Model. We will call this variant the "Argonne-SRI Model". In this model, each processor has a private binding array just as in the SRI Model. However, the binding array only contains unfavoured bindings. Favoured bindings are implemented as in the Argonne Model, by assigning to the variable value cell with a special mark.

On switching of tasks, the binding array is changed in exactly the same way as in the SRI Model. This will result in a set of unfavoured bindings being removed from the binding array, and a new set of unfavoured bindings being installed. No favoured bindings will be involved, since we assume a processor doesn't switch tasks until it has backtracked to the node from which it started its current task, by which point it will have undone any favoured bindings it has made.

When the favoured processor at a node backtracks past the node leaving other processors still working beneath the node, it is in principle possible for one of the

[2]This follows from our assumption that idle processors pick nodes as high up the tree as possible.

other processors to be promoted to be the favoured processor at that node. The promoted processor needs to check all the bindings in its private binding array and remove any which have become favoured, transferring the binding to the variable value cell. A similar possibility is available in the Manchester-Argonne Model. For either model, it seems doubtful whether the cost of performing the promotion is worthwhile.

The Argonne-SRI Model's operational behaviour is very similar to the SRI Model, in that it takes constant time to access or bind a variable, but switching tasks is relatively expensive. The cost of switching tasks is proportional to the number of variable bindings on the path between the nodes (as in the SRI Model).

In implementation terms, what the Argonne-SRI Model does within a task is remarkably close to the standard sequential model. It is only in the accessing and binding of unfavoured variables that significant extra time or space is required. The SRI Model, on the other hand, incurs overheads on most variable accesses. There will be an extra memory access to the binding array, and the binding array needs to be large enough to store the majority of variable bindings, rather than just those that are unfavoured. For these reasons, the Argonne-SRI Model is probably preferable to the SRI Model.

8 Conclusion

Which of these models is best? We feel it is very difficult to reach a firm conclusion until we are in a position to do a detailed experimental comparison of the models on large-scale programs. A lot depends on the strategy used for selecting which branches to execute in or-parallel, and the user annotation which guides this.

Our current feeling of the order of merit as practical models is:

```
                          Argonne < Manchester-Argonne
Abstract < Naive <
                          SRI < Argonne-SRI
```

Thus our choice would be between the Argonne-SRI and Manchester-Argonne models. Both models have the advantage of being virtually identical to a standard sequential model except for the treatment of unfavoured variables.

Of the two, the Argonne-SRI Model is somewhat simpler, and guarantees that all variable accesses (even unfavoured) are very fast, constant-time operations. Its main disadvantage is the cost of switching tasks. Thus it requires a selection strategy that

avoids generating very small tasks. If such a strategy can be achieved, it should perform very well. [3]

If more fine-grained parallelism is unavoidable, then the Manchester-Argonne Model seems the best choice. It offers a good compromise between speed of access to variables and speed of switching tasks. Although access to unfavoured variables is relatively expensive, the cost is kept bounded. In this respect it seems to offer a significant improvement over the Argonne Model.

Results by Kish Shen [5], on simulating relatively small-scale examples, show that the percentage of variable reference that are unfavoured can often be uncomfortably high. Greater than 20% is common and greater than 50% is not unusual. [4] A good selection strategy can usually keep the percentage low, but such a strategy will generally favour an SRI-like model in any case. If we cannot rely on a good selection strategy, and need an Argonne-like model, then it seems vital to keep down the cost of unfavoured variable accesses, which is what the Manchester-Argonne Model achieves.

9 Acknowledgements

The ideas described in this paper owe much to interactions with colleagues at Argonne National Laboratory, the Swedish Institute of Computer Science, and Manchester University. The presentation benefitted particularly from comments by Mats Carlsson, Ross Overbeek and Kish Shen.

The work was supported by the UK Science and Engineering Research Council.

References

[1] R. Butler, E. L. Lusk, R. Olson, and R. A. Overbeek. *ANLWAM: A Parallel Implementation of the Warren Abstract Machine.* Internal Report, Argonne National Laboratory, U.S.A, 1986.

[2] J. Gabriel, T. Lindholm, E. L. Lusk, and R. A. Overbeek. *A Tutorial on the Warren Abstract Machine.* Technical Report ANL-84-84, Argonne National Laboratory, Argonne, Illinois, Oct. 1984.

[3] T. Miyazaki, A. Takeuchi, and T. Chikayama. A sequential implementation of Concurrent Prolog based on the shallow binding scheme. In *The 1985 International Symposium on Logic Programming*, pages 110–118, IEEE, 1985.

[3] A possible way to achieve this, suggested by Kish Shen, is to prevent an or-parallel node from being shared until the processor that generated it has performed some critical number of further steps beneath that node.

[4] These figures typically refer to cases where there are no restrictions on which nodes can be executed in parallel. A more restrictive annotation generally brings the percentages down.

[4] R. A. Overbeek, J. Gabriel, T. Lindholm, and E. L. Lusk. *Prolog on Multiprocessors.* Internal Report, Argonne National Laboratory, U.S.A., 1985.

[5] K. Shen. *An Investigation of the Argonne Model of Or-Parallel Prolog.* Master's thesis, University of Manchester, 1986.

[6] J. Syre and H. Westphal. *A Review of Parallel Models for Logic Programming Languages.* Technical Report CA-07, ECRC, 1985.

[7] E. Tick and D. H. D. Warren. Towards a pipelined Prolog processor. *New Generation Computing*, 2(4):323–345, 1984.

[8] D. H. D. Warren. *An Abstract Prolog Instruction Set.* Technical Note 309, SRI International, 1983.

[9] D. S. Warren. Efficient Prolog memory management for flexible control strategies. In *The 1984 International Symposium on Logic Programming*, pages 198–202, IEEE, 1984.

A Appendix

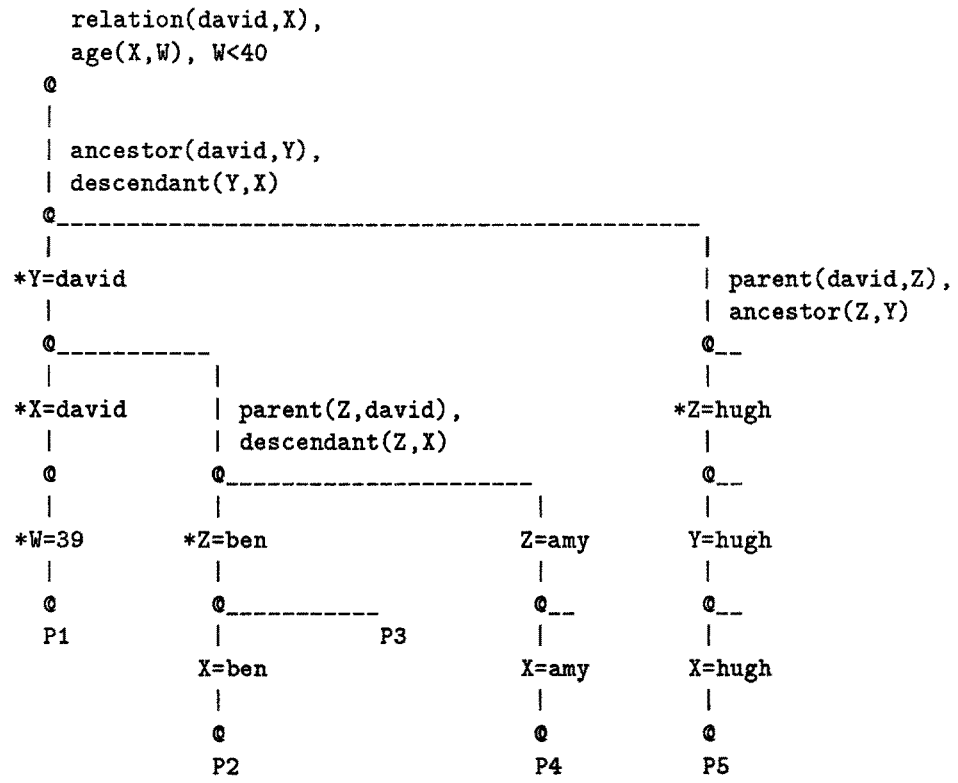
In this appendix, we illustrate some of the main features of the different models. The following program is used as an example.

A.1 Example Program

```
relation(X,Z) :- ancestor(X,Y), descendant(Y,Z).

ancestor(X,X).
ancestor(X,Z) :- parent(X,Y), ancestor(Y,Z).

descendant(X,X).
descendant(X,Z) :- parent(Y,X), descendant(Y,Z).

parent(amy,david). age(amy,7).
parent(ben,david). age(ben,9).
parent(david,hugh). age(david,39).
parent(david,winifred). ...
...

?- relation(david,X), age(X,W), W < 40.
```

A.2 Search Tree

This is a particular state of the search tree for the query given in the example program. Five processors are active, and the tasks they are currently working on are shown below the tree. Each node of the tree is marked with the new goals introduced that were introduced at that node; the new bindings introduced are shown on the arc above. Bindings that are favoured in the Argonne-style models are marked with an asterisk. Unexplored alternatives are indicated with a short horizontal arc.

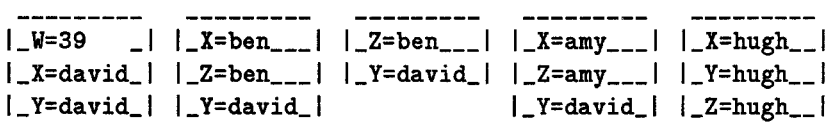


Tasks

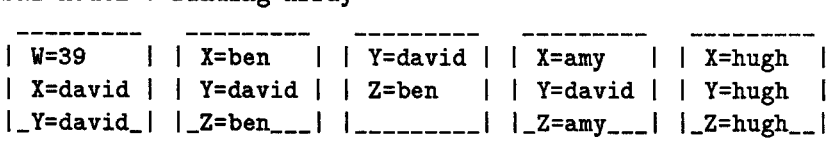
39<40	age(ben,W), W<40	desc(ben,X), age(X,W),	age(amy,W), W<40	age(hugh,W), W<40

This is how the different models record the binding information for each of the five processors. A box denotes an element of the binding list in the Naive Model, or a hash table in the other models. Chains of hash tables are indicated in an obvious way, with the first table to be searched topmost.

Naive Model : Binding List

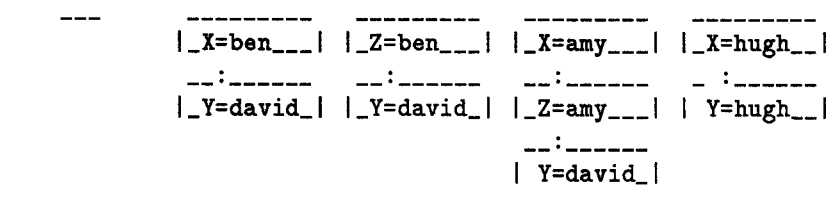


SRI Model : Binding Array

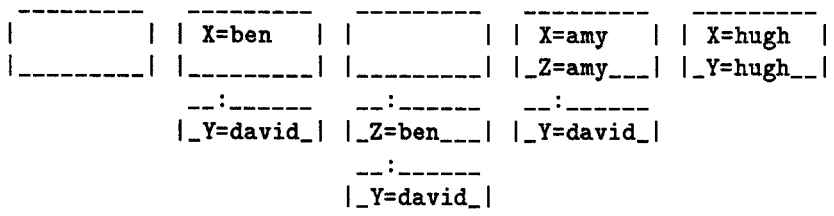


Argonne-SRI Model : Binding Array

Argonne Model : Relevant Hash Table Chain

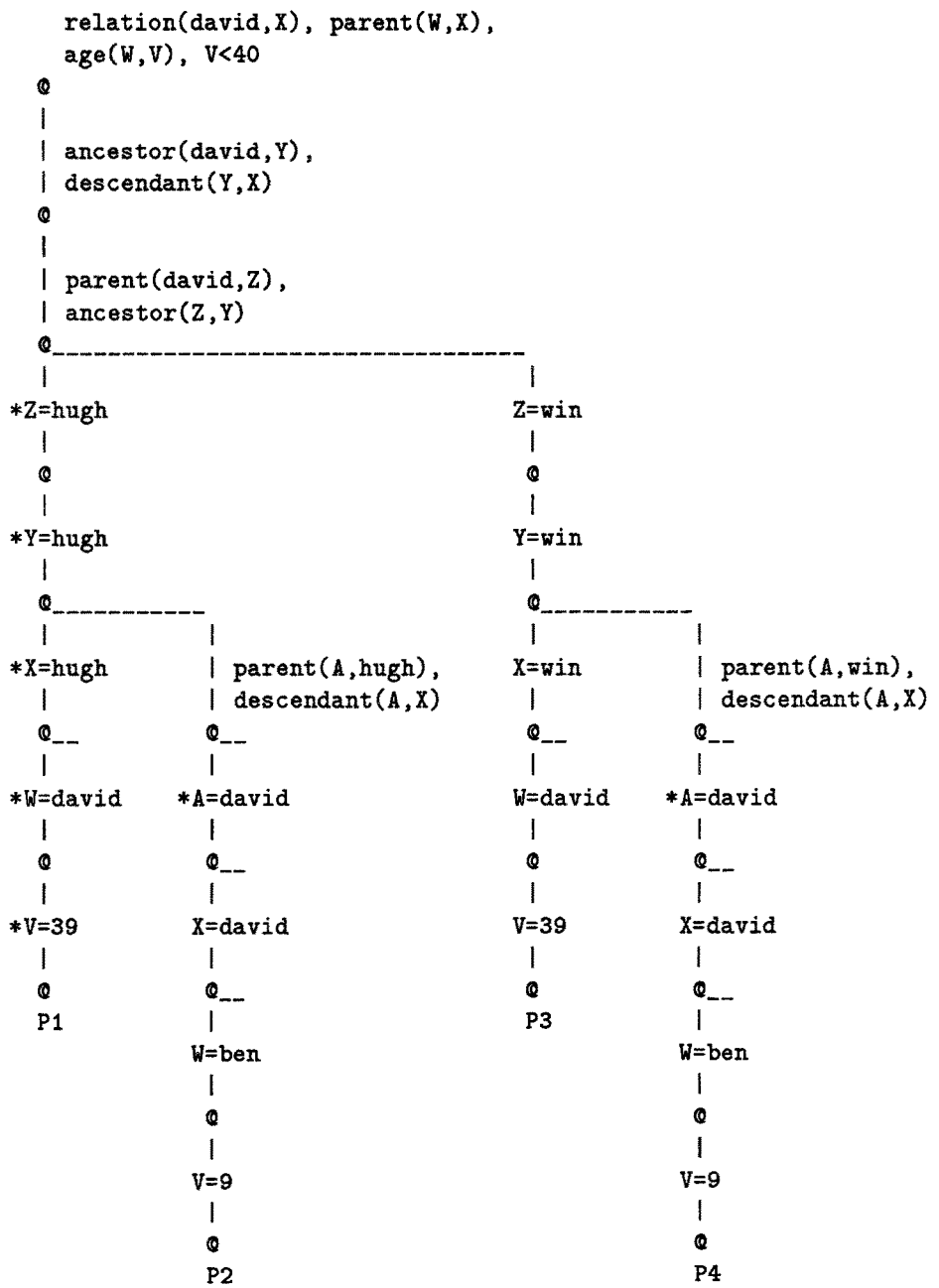


Manchester-Argonne Model : Binding Array + Hash Table Chain



A.3 Second Search Tree

Here is another search tree, for a somewhat different query. Four processors are active.



Here is the binding information for the Argonne Model versus the Manchester-Argonne Model.

```
Argonne Model : Relevant Hash Table Chain

    ---      ---------                ---------   ---------
            | V=9     |              | V=39    | | V=39    |
            |_W=ben___|              |_W=david_| |_W=ben___|
             __:______               __:______   _ :______
            |_X=david_|              |_X=win___| | X=david_|
             __:______               __:______   __:______
            |_Y=hugh__|              |_Y=win___| |_Y=win___|
             __:______               __:______   __:______
            |_Z=hugh__|              |_Z=win___| |_Z=win___|

Manchester-Argonne Model : Binding Array + Hash Table Chain

 ---------   ---------                ---------   ---------
|         | | V=9     |              | V=39    | | V=39    |
|         | | W=ben   |              | W=david | | W=ben   |
|         | | X=david |              | X=win   | | X=david |
|         | |         |              | Y=win   | |         |
|_________| |_________|              |_Z=win___| |_________|
             __:______                            __:______
            | Y=hugh  |                          | Y=win   |
            |_Z=hugh__|                          |_Z=win___|
```