

Chapter 13

Restricted And-Parallelism and Side-Effects in Logic Programming

Doug DeGroot
Computer Science Center
Texas Instruments

13.1 Introduction

With the increasing reliance upon artificial intelligence (AI) to solve many of today's complex military, industrial, social, and business computer applications, the demands for ever higher performance in the area of symbolic computing continue to mount. More and more, researchers are looking for methods of executing AI languages in parallel. The majority of this attention has focused on models of parallel Lisp [Halstead 88] and parallel Prolog [Conery 81, Wise 86].

Several novel approaches have been developed recently for executing logic programming languages, such as Prolog, in parallel. (Prolog is probably the most well-known logic programming language.) Although several approaches are being pursued, the majority involve variations of two basic mechanisms: *and-parallelism* and *or-parallelism* [Conery 81]. Although Prolog is the most successful and most-widespread logic programming language, many approaches to parallel logic programming have either abandoned the semantics of Prolog or have developed new language extensions or even new languages. Examples of these approaches include Concurrent Prolog [Shapiro 83], Parlog [Clark 86], GHC [Ueda 86], and Epilog [Wise 86].

Conery's model is a notable exception [Conery 87]. In his *And/Or Process* model, a set of run-time tests and algorithms are executed in order to derive a run-time ordering of goals within a Prolog clause and to produce a possibly-parallel execution graph based on this ordering. Although Conery's approach does retain the original Prolog syntax and semantics, it does so at a fairly considerable expense. A simpler and cheaper model was presented in [DeGroot 84]. This simpler model, called the *Restricted And-Parallelism (RAP)* model, also retains the original syntax and semantics of Prolog, but with significantly less run-time expense. The run-time expense is reduced through a set of approximation techniques which make certain assumptions on the possible execution traces of a clause. Because the techniques used are approximation techniques, they may in fact occasionally result in less parallelism than possible. Conery's model does not suffer from this fault; his approach remains the standard by which maximal parallelism can be achieved.

The RAP model trades maximal detection of parallelism for more efficient execution. As explained in [DeGroot 84], it is believed that the RAP model will detect more than enough parallelism to keep a moderately-sized parallel processor system busy. Recent empirical evidence certainly lends support to this thesis [Carlton 88]. Techniques may in fact be needed to limit the run-time detection of parallelism of RAP; consequently, the execution model of RAP allows parallelism-extraction code sequences to execute as if they were sequential code sequences [DeGroot 84].

The present paper describes the RAP model, and in particular, shows how it can efficiently support side-effect computations [DeGroot 87].

13.2 Parallel Execution of Prolog Programs

Many excellent introductory texts on logic programming and Prolog exist [Lloyd 84, Shapiro 86, Kluzniak 84], and so only a minimal description is provided here. The goal of the description is to focus on the execution semantics of Prolog rather than the language features, so that the parallel execution model presented below can better be appreciated.

First, Prolog defines only three types of statements: facts, rules, and queries. All three statement types are called *clauses*. Examples of each type of clause are shown below:

Facts:

```
isa(fido,dog).
father(john,mary).
part(partno(p2),pname(bolt),color(green),weight(17)).
```

Rules:

```
grandparent(X,Y) :- parent(X,Z), parent(Z,Y).
happy(D) :- isa(D,dog), has_a(D,X), isa(X,bone).
```

Queries:

```
?- grandparent(sue,C).
?- country(C), borders(C,mediterranean), country(C1),
   asian(C1), borders(C,C1).
```

Facts constitute the fundamental "truths" of the program. Each fact asserts that a certain specific relation holds between its arguments; the meaning of the relation is implicit except for certain predefined system relations. In the above, constants begin with lowercase letters; they refer to specific individual elements of the name space. Variable identifiers begin with uppercase letters and are universally quantified [Lloyd 84]. Variables may assume any value whatsoever, including the value of another variable. When this occurs, aliases may arise.

Rules express conditional "truths". Each rule specifies a conditional *conclusion* to the left of the arrow (i.e., the ":-") and one or more *preconditions* to the right of the arrow. The conclusion is frequently called the *goal* (or *head*) of the clause, and the preconditions are called the *subgoals* (or *body*). To determine the truth of the conclusion of a rule, a Prolog system must recursively establish the truth of each of the preconditions in the rule.

Queries are the language mechanism used to activate the proof procedure on rules. As can be seen in the above examples, queries also contain one or more subgoals. Every subgoal in a query must be solved for the query to be solved (or, proven).

The data structures appearing as arguments in goals, subgoals, and facts are called *terms*. Terms can be arbitrarily structured; they are defined inductively as follows:

1. A variable is a term.
2. An atomic element is a term (e.g., constants, integers, characters).
3. If f is an alphanumeric symbol (or more precisely, a *functor* [Lloyd 84]), and t_1 through t_n are terms, then $f(t_1, t_2, \dots, t_n)$ is also a term.

Normal Prolog execution involves the sequential, left-to-right execution of the subgoals in a clause or query. A subgoal can be solved by finding a "matching" fact or by finding a rule whose head "matches" and whose body can be recursively solved, also in a left-to-right manner. The matching process referred to is called *unification* [Lloyd 84]. It is a recursive process in which each argument (*term*) of a subgoal is matched with the corresponding argument in the same position of the fact or rule head with which the match is to be attempted. Unification operates on two terms. The two terms unify if they are identical atomic symbols, if either or both is a variable, or if the terms have the same functor name and the same number of arguments and if these arguments are themselves recursively unifiable. During unification, assignment occurs if one or both of the terms to be unified is a variable. In such cases, one variable is assigned the value of the other term. The variable assigned the value of the other term can be either in the subgoal or in the fact or rule head. There is thus two-way assignment. The assignment is by reference and not by value; thus aliases can easily arise.

Every successful execution of a query results in an and-execution tree, as illustrated in Figure 13.1.¹ In these trees, leaf nodes represent Prolog facts; internal nodes represent rules, where the node is labeled with the head of the rule represented by the node. In general, a query may be able to be solved in more than one way; each solution produces a possibly unique and-execution tree. Figure 13.1 shows two possible and-execution trees for the single query. And-parallelism in logic programs is concerned with finding techniques to execute the nodes of the and-execution tree in parallel, both internal and leaf nodes, rather than in the traditional, sequential manner.

--- figure 13.1 here

Several problems arise when this is attempted. First, care must be taken to ensure that the values produced for variables are consistent among all executed goals. For example, consider the following simple program and query:

```
parent(bob,sally).  
parent(jim,sue).
```

¹Query execution can also be considered as producing and/or execution trees. These are of interest mainly when considering or-parallel execution models [Crammond 85] or combined and-or execution models, as in [Conery 87]. In this paper, only and-parallel execution models are considered, and so only and-execution trees are of interest.

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

?- grandparent(bob,sue).

When the *grandparent(bob,sue)* query is executed, it matches the *grandparent* rule, giving the value of *bob* to the variable *X* and the value *sue* to the variable *Y*. To solve the grandparent rule then, the two subgoals must be solved with the assigned variable values. The two modified subgoals are thus *parent(bob,Z)* and *parent(Z,sue)*. Suppose an attempt is made to execute these two subgoals in parallel. The first subgoal may succeed by matching the fact *parent(bob,sally)*, thereby assigning the value *sally* to the variable *Z*; the second subgoal may succeed by matching the fact *parent(jim,sue)*, thereby assigning the value *sue* to the variable *Z*.

Because both subgoals have succeeded, it is tempting to conclude that the entire rule has succeeded. But note that the two subgoals succeeded only by assigning two different values to the single variable *Z*. No single value of *Z* has been found such that *bob* is the parent of *Z* and *sue* is the child of *Z*. Consequently, the rule has not been successfully executed, and we do not know if *bob* is a grandparent of *sue*. When two different values are assigned to a single value, a *binding conflict* is said to have occurred.

Several techniques have been proposed to prevent this problem. One obvious technique is to have each subgoal produce all possible answers instead of just one and to then compute the relational join over the variables shared by the subgoals in the rule [DeGroot 88b]. This technique is examined in detail in [Pollard 81]. If the empty set results from the join operation, no answer can be produced for the rule, and the rule fails. If the answer set is non-null, this set is returned as the answer to the rule. Since the rule may have been invoked by some other rule, the answer set may be passed up the and-execution tree as the answer to the rule's parent subgoal, helping constitute the answer for the parent rule.

As pointed out by Conery, this technique exhibits a few fundamental problems, one of which occurs simply when executing built-in arithmetic predicates and side-effect procedures that rely on Prolog's traditional sequential data flow [Conery 87]. This technique also suffers from certain performance anomalies which may make its parallel performance worse than sequential performance in cases when only one answer to a query is desired instead of all answers.

Several other approaches to and-parallel execution of logic programs are currently being investigated; each exhibits significantly different performance characteristics, and no single model has yet emerged as "best" [Conery 87, DeGroot 88b, Wise 86].

13.3 Overview of the RAP Model

In this section, another technique for and-parallel execution of logic programs is presented. This approach is called Restricted And-Parallelism (RAP); it provides an efficient means of executing traditional Prolog programs in parallel while retaining the full semantics of sequential Prolog and hopefully offering the efficiency of single-answer execution [DeGroot 84]. The essential concept of the model is to compute at compile-time a single execution graph expression which can at run-time result in multiple, different parallel execution behaviors, and to do so with minimal run-time support. To do so, it must monitor during run-time all potentially executable subgoals and ensure that no two subgoals execute in parallel if they share one or more unbound variables. If this were allowed, the two subgoals might attempt to

assign different values to the shared, unbound variable(s), and a binding conflict would have occurred.² This approach is exactly opposite that studied by Pollard.

The three main components of the RAP model are 1) the typing algorithm, 2) the independence algorithm, and 3) the execution graph expressions. Each of these is described below.

13.3.1 The Typing Algorithm

The *typing algorithm* is responsible for determining and maintaining the types of all terms (the data objects of Prolog) in an attempt to maintain an accurate indication of whether the term contains any unbound variables (unbound pointers) within it. A term may be of type *variable* (the dereferenced pointer value is unbound), in which case it is assigned the type V; it may be a *ground term*, a term containing no unbound variables, in which case it is assigned the type G; or it may be a *non-ground, non-variable* term, in which case it is assigned the type N. The terms are typed recursively so that all terms and components of terms are typed. Figure 13.2 illustrates some simple examples of terms and their types.

-- figure 13.2 here --

During execution of a Prolog program, terms of type V (variables) may be bound to other terms of type V, G, or N; when variables become bound, they assume the type of the term to which they are bound. Terms of type N may become ground, and therefore of type G, by having their internal unbound variables bound to ground values. Terms of type G cannot change types (except through backtracking [Hermenegildo 87]).

Several possible typing algorithms are possible. A run-time checking algorithm was first described in [DeGroot 84]. This algorithm attempts to lower the run-time overhead by making only a partial, top-level check of a term's contents instead of checking the term in depth. As a result of this partial check, a term may occasionally be typed too strongly, that is, it may be typed as non-variable, non-ground when in fact, it may be ground. To make an absolute determination of the type of a term, it would frequently be necessary to perform during run-time a complete scan of the term to see if it contains any unbound variables. Because the term may be arbitrarily large, run-time scans are prohibitively expensive, especially when it is considered that in the worst case, these scans may occur following every successful subgoal invocation. The approximation to the typing is a compromise between run-time overhead and the need to make accurate type determinations.

Another typing algorithm based on a static data-dependency analysis is possible which incurs even less run-time overhead but which may occasionally have poorer performance [Chang 85, DeGroot 85]. The approach computes a set of possible activation "modes" for each clause and then settles for the worst case mode in assigning types. In certain cases, this algorithm may make an accurate assessment of a term's type throughout the entire program execution, but in others, it may significantly underestimate the state of a term.

A very simple typing algorithm could simply use the terms' normal WAM type codes

²In general, it is difficult to know for sure whether a binding conflict will occur, so the RAP model takes the conservative approach of simply assuming one will and therefore prevents parallel execution of such goals.

[Warren 83] - variable, constant, list, or structure. Variables would be assumed (naturally) to be of type V, constants would be type G, and lists and structures can simply be assumed to be of type N. No further typing overhead would then be required. The WAM's normal data type code assignments would be used to compute a RAP independence type code. Additional compiler directives, such as mode declarations or asserting that all lists contain only independent elements, would allow improved typing assumptions [DeGroot 88a]. The important point to note here is that all overhead can be eliminated from the typing algorithm component of the RAP model.

Other typing algorithms are clearly possible. The RAP model is defined independently of any particular typing algorithm.

13.3.2 The Independence Algorithm

The *independence algorithm* is responsible for determining when two terms are independent, that is, when neither shares an unbound variable with the other. The RAP execution model uses the independence algorithm to determine when two subgoals are independent: if two subgoals are invoked with arguments (terms) which are all independent from each other's, then the two subgoals are independent and may potentially execute in parallel; otherwise, they must execute sequentially in order to ensure that no binding-conflicts occur. Figure 13.3 illustrates the independence algorithm.

---Figure 13.3 here ---

This independence constraint is sufficient but not necessary to ensure correct parallel execution: just because two subgoals are not independent, it cannot be concluded that a binding conflict *will* occur if they execute in parallel, only that one *might* occur. The RAP model is perhaps overly conservative in this respect. Significant opportunities exist for an optimizing compiler to assist in determining when this constraint can be relaxed [DeGroot 88a]

13.3.3 The Execution Graph Expressions

The *execution graph expressions* are used to express the potential parallel execution sequences of the subgoals in a clause. Six expression types were originally defined, although others are possible. They are defined recursively as follows:

1. G
a simple goal (or subgoal); this is the simplest expression type.
2. (SEQ E1 . . . En)
Execute expressions E1 through En sequentially.
3. (PAR E1 . . . En)
Execute expressions E1 through En in parallel.
4. (GPAR(V1...Vk) E1 . . . En)
If all the variables V1 through Vk are ground (have type G), then execute expressions E1 through En in parallel; otherwise, execute E1 through En sequentially.
5. (IPAR(V1...Vk)(Vm ... Vp) E1 . . . En)

If each variable V_1 through V_k is independent from every variable V_m through V_p , then execute expressions E_1 through E_n in parallel; otherwise, execute E_1 through E_n sequentially.

6. (IF B E_1 E_2)

If expression B evaluates to true, execute expression E_1 ; otherwise, execute expression E_2 .

The GPAR test simply examines the type fields of the specified terms to see if they are ground. The IPAR test simply invokes the independence algorithm on all pairs of variables in the IPAR expression; generally, the number of tests required is actually very small [DeGroot 88a].

13.4 Examples of Execution Graph Expressions

As examples of the types of execution graph expressions generated for typical Prolog clauses, consider the following clause:

$f(X) :- g(X), h(X), k(X).$

This clause may be compiled into the following execution graph expression:

$f(X) :- (GPAR(X) \ g(X) \ (GPAR(X) \ h(X) \ k(X)) \).$

This single execution graph expression can result in any of the three run-time execution graphs of Figure 13.4. For example, if f is invoked with a ground value of X , as in "f(dog)", then the first GPAR test succeeds, thereby activating the two subexpressions in parallel. The first subexpression activates the subgoal $g(X)$. The second subexpression simultaneously begins execution since the successful GPAR test guarantees that no binding conflicts can occur. This second subexpression also contains a GPAR test; it tests X again, finds it to be ground, and activates the two additional subexpressions in parallel.³ Thus $h(X)$ and $k(X)$ begin parallel execution along with $g(X)$. The resulting execution graph is shown in Figure 13.4.a. The GPAR tests assure that no binding conflicts will occur even though all three subgoals are executing in parallel with the same term.

Suppose now that the first GPAR test fails. Clearly then X is not ground and so contains an unbound variable somewhere within it. Consequently, no two of the subgoals can be allowed to execute in parallel, as otherwise a binding conflict might occur. Since the GPAR test failed, the two subexpressions must execute sequentially. So the first subgoal, $g(X)$, begins execution alone. When and if it successfully completes, the second subexpression can begin execution. Because the execution of $g(X)$ may have resulted in X 's becoming ground, the second subexpression retests X . If X is now ground, the GPAR test succeeds, and the two subgoals $h(X)$ and $k(X)$ can execute in parallel; the resulting run-time execution graph is shown in Figure 13.4.b. If the second GPAR test fails, X must still be non-ground, and so $h(X)$ and $k(X)$ must execute sequentially, resulting in the run-time execution graph of Figure 13.4.c.

-- figure 13.4 here --

³Note that this second GPAR test cannot fail. Since X was ground in the first test, it must still be ground. Terms of type G cannot change their types except through backtracking.

Note that three different run-time execution graphs are possible for the example Prolog clause but that only one RAP execution graph expression is needed to express all three execution graphs, and this is computed at compile-time. Simple run-time tests are used to dynamically select one of a number of possible execution graphs given the execution graph expression. This is a significant advantage of the RAP model.

For another example, consider the clause:

```
f(X,Y) :- p(X), q(Y), s(X,Y), t(Y).
```

This clause may be compiled into the execution graph expression:

```
f(X,Y) :- (GPAR(X,Y)
           (IPAR(X,Y) p(X) q(Y))
           (GPAR(Y) s(X,Y) t(Y)) )
```

or even into:

```
f(X,Y) :- (GPAR(X) p(X)
           (GPAR(Y) q(Y)
            (GPAR(Y) s(X,Y) t(Y))))).
```

The first execution graph expression above can result in a slightly different set of run-time execution graphs than the second; it is interesting to compare the two.

The graph expression tests can be combined when necessary. For example, consider the clause:

```
f(X,Y,Z) :- g(X,Y), h(Y,Z).
```

This clause can be compiled as:

```
f(X,Y,Z) :- (GPAR(Y) IPAR(X,Z) g(X,Y) h(Y,Z)).
```

Here, both the GPAR and IPAR tests must succeed for the two subgoals to execute in parallel.

It should be noted that the run-time tests are very simple and efficient. They can easily be implemented in hardware or firmware. Manuel Hermenegildo has in fact produced an extension to Warren's Abstract Machine for Prolog (the WAM [Warren 83]) that incorporates direct hardware support for the RAP model. The RAP graph expressions are easily compiled into this extended WAM model, and a sophisticated run-time kernel supports efficient parallel execution of the compiled Prolog program [Hermenegildo 87].

13.5 Backtracking in the RAP Model

To consider how normal Prolog backtracking can be implemented in the RAP execution model, a technique presented by Hermenegildo is described here [Hermenegildo 87]. Although it is not strictly necessary to understand backtracking to understand the following extension of the RAP model to include side-effects, it is important. This section may however be skipped.

First, it is important to consider how Prolog handles multiple clauses in a procedure. For example, consider the following simple procedure:

```
p :- a, b, c.
p :- g, h.
p :- q, r, s.
```

If we are given the goal `p` to solve, there are three different clauses that can be used in the proof. Or-parallel execution techniques proceed by attempting all three clauses in parallel; if

any clause leads to success, then the query succeeds [Crammond 85]. And-parallel execution, however, attempts only one clause at a time; but within the selected clause all subgoals may potentially execute in parallel. If the selected clause leads to failure, execution backs up to the most recent, previous execution point (subgoal) where untried clauses remain, and another clause is selected. If none of these remaining clauses lead to success, the next most recent execution point with remaining subgoals is examined, and so on. This process is called *backtracking*.

In the Warren Abstract Machine, Prolog execution maintains a run-time procedure-call stack. This stack contains procedure return addresses, actual parameter values, and other control information [Warren 83]. In particular, the stack contains *choice points* to mark those procedures that still contain one or more untried clauses. These choice points are used in backtracking. When and-parallel execution is involved, the notion backtracking to a "previous" subgoal is more complex. There is no chronological ordering that can be relied upon to implement normal backtracking. Consequently, additional information must be maintained in the stack.

In Hermenegildo's system, choice points are called *choice point markers* (CPMs). An additional marker type is defined called *parallel call markers* (PCMs). Parallel call markers are used to record those points where a RAP execution graph expression resulted in the parallel execution of two or more subexpressions. As execution proceeds, CPMs and PCMs are recorded in the execution stack to record the execution points where clause choices were made and where parallel execution occurred. The backtracking algorithm can then be implemented as follows:

- During normal, forward execution, record choice point markers (CPMs) in the stack whenever a subgoal execution contains one or more untried program clauses. Record a parallel call marker (PCM) in the stack whenever a conditional graph expression evaluates to true and produces parallel subexpressions. Mark each PCM as "inside" when it is created; change its value to "outside" if all subexpressions within the associated graph expression succeed.
- If a subgoal fails, find the last (topmost) marker in the stack.
 - If it is a CPM, all execution since that choice point has been sequential, so backtrack normally to that point and try one of the remaining untried clauses (as in sequential execution).
 - If it is a PCM and its value is "inside", then several expressions are executing in parallel. Furthermore, at least one of these expressions has not yet completed. Cancel all unfinished subexpressions within the PCM's associated graph expression, and recursively fail back (i.e., go to the previous marker).
 - If it is a PCM and its value is "outside", then several expressions were executing in parallel, but all have finished. Find the last subexpression (using source-code textual order) that has remaining alternative clauses (represented by its associated choice point marker). Try these untried clauses one by one until one is found that succeeds; then restart in parallel all textually following expressions in normal, forward execution mode. If none of the untried clauses leads to success, fail back (i.e., go to the previous marker).

This simple, extended backtracking algorithm for RAP is easily implemented in Hermenegildo's extended abstract machine.

13.6 Side-Effects

Unlike pure logic, logic programming languages such as Prolog have the curious aspect of occasionally having side-effects as a result of and during execution of a logic program. Goals, subgoals, and predicates that produce side-effects include those such as *assert* and *retract*, *read* and *write*, *cut*, and some other commonly used side-effect producing procedures [Shapiro 86]. In addition, in the RAP model, goals which can produce possible machine-checks or system faults are also considered to be side-effect goals. Examples include *divide*, which might attempt to divide by zero and cause a program trap, and those that require a certain threshold in terms of the number of instantiated arguments, again, such as *divide*, which is required to have its first two arguments instantiated to non-variable values, and arithmetic comparison predicates, such as *gt*, *lt*, and so on. Other examples include *var*, *integer*, *plus*, *is*, and the like. All of these are classified as side-effect goals within the RAP model.

The particular operational semantics of Prolog involve a simple, sequential execution model in which subgoals within a clause are tried in order, from left to right. In addition, multiple clauses within a single procedure are tried one at a time, from top to bottom (at least, within the domain of the selected, indexed clauses [Warren 83]). These two orderings, when coupled with the observed side-effects, produce a given program's observable behavior. To execute a Prolog program in parallel and retain its normal Prolog-defined semantics, it is necessary to retain the order of observable side-effects in the program. As long as this order is retained, the goals themselves may execute in any order. Thus it is not really the left-to-right and top-to-bottom execution orderings of all goals and clauses that give Prolog its unique semantics, it is really the induced ordering of the side-effects produced. If, for example, a clause contains only subgoals which are side-effect free, then the order of execution of these subgoals is irrelevant, and they may potentially execute in parallel. (Performance may differ, but the produced results will be identical [Warren 80].)

The semantics of Prolog then do not really require a sequential logic programming execution model, they only appear to with respect to the sequence of observable side-effects. The side-effect goals must retain their left-to-right and top-to-bottom, induced ordering; but all other goals are, to a certain extent (as explained below), order-independent. In fact, the goals in a side-effect free Prolog program can be executed in any order whatsoever. It is only when side-effect goals are introduced that order becomes important. (Order independence should not be confused with the ability to execute in parallel.)

13.6.1 Side-Effect Goals - Some Terminology

For ease of discussion throughout the remainder of this chapter, an informal terminology is introduced. First, as previously mentioned there is a certain set of predefined, evaluable predicates in Prolog which when executed result in (or may result in) certain side-effects to the normal execution. This set includes such evaluable predicates as *assert* and *retract*, *read* and *write*, *cut*, and so on. These predicates are referred to below as **side-effect built-ins**. A clause that contains a side-effect built-in is called a **side-effect-clause**, and a procedure that contains at least one side-effect-clause is called a **side-effect procedure**. A subgoal in a clause that calls

a side-effect procedure is considered a side-effect goal, although it is more appropriately called a **potential side-effect goal**. Any clause containing such a potential side-effect goal is called a **potential side-effect clause**, or simply, a **side-effect clause**. A goal or subgoal that is neither a side-effect goal nor a potential side-effect goal is called a **pure goal** or **pure subgoal**. A clause that contains only pure subgoals is called a **pure clause**, and a procedure containing only pure clauses is a **pure procedure**. Clearly, a (potential) side-effect procedure may contain both side-effect clauses and pure clauses, and so may or may not produce a side-effect when invoked. A potential side-effect goal may unify with the heads of both side-effect clauses and pure clauses, but a pure goal may call only pure procedures and thus may unify with only the heads of pure clauses.

A compiler can easily make a recursive traversal of the program source code and determine the type of every goal, clause, and procedure with respect to side effects. The required algorithm is similar to Mellish's automatic mode detection [Mellish 81] and to the static data-dependency analysis traversal algorithm of [Chang 87] (see also [O'Keefe 87]). Once the compiler has ascertained the type of each goal, clause, and procedure, the required synchronization code can be added to the RAP execution graph expressions, as described below.

13.6.2 Sequencing Side-Effect Goals

As described in Section 13.2, successful execution of a Prolog query produces an and-execution tree in which inner nodes represent rules and leaves represent either simple Prolog facts or built-in Prolog predicates. The successors of an internal node represent the subgoals of the rule. The leaves of a tree, when considered as a list, in order, from left to right, represent all final subgoals that had to eventually be proven in order to prove the initial query. In Figure 13.1.b, this list is (a,b,c,d,e,f).

In general, consider the list of leaves in an execution tree that results from proving a query. If all the goals represented by these leaves are pure, then they can execute in any order. They may also possibly be executed simultaneously with full and-parallelism (if they are all independent), or perhaps at least in any order using partial and-parallelism (if they are at least partially independent). In any event, it is clear that the order of execution and the degree of parallelism is irrelevant with respect to the program's observable behavior since the program has no observable behavior except in the top-level reply to the query. Whether the program is executed in order or out of order, sequentially or with restricted and-parallelism, the result is the same.

Consider now that some of the leaves represent side-effect goals, such as *write* predicates that write to a user terminal. Clearly, these *write* predicates must execute in their original, left-to-right, top-to-bottom order. If not, the resulting output stream might not appear the same as when the program is executed sequentially. So the *write* predicates must still execute in order. Since the remainder of the goals are pure goals, it might appear that they could execute in any order; but this is not so, as can be seen by considering the following example clause:

```
p(X) :- test(X), write(X).
```

It is tempting to compile this clause as:

```
p(X) :- (GPAR(X) test(X) write(X)).
```

But this would be wrong. On entry, if *X* is not ground, then the two subgoals execute sequentially, as in normal Prolog execution, and no problems can arise. But note that here, the

first subgoal is intended to execute successfully before the *write* subgoal executes. If it fails, the *write* subgoal must not execute, for the write subgoal will exert an irrevocable side-effect upon the program's observable behavior. If *X* is ground on entry, the GPAR test succeeds, and the two subgoals can execute in parallel; the *write* subgoal may precede the execution of the *test* subgoal, thereby leading to possible errors. This clause would then be better compiled as:

$p(X) :- (\text{SEQ } \text{test}(X) \text{ write}(X)).$

It can be seen by extension that all subgoals preceding a side-effect goal must execute successfully before the side-effect goal can execute (at least those that may potentially fail). What about pure goals following a side-effect goal? Can they execute in any order? Consider the clause:

$p(X) :- \text{test}(X), \text{write}(X), m(X).$

Assume *m* is pure. Then if *X* is ground, *m(X)* can execute in any order whatsoever, even before either of the first two goals. Because *m(X)* has no side-effects, it cannot affect the execution in any way whatsoever. Thus even if *test(X)* fails, *m* can still execute without changing the set of producible answers and observable behaviors. So this clause can be compiled as:

$p :- (\text{SEQ } \text{test}(X) (\text{GPAR}(X) \text{ write}(X) m(X))).$

In fact, since it is certain that the *write* subgoal cannot affect the value of *X*, the clause can even be compiled as follows:

$p :- (\text{SEQ } \text{test}(X) (\text{PAR } \text{write}(X) m(X))).$

Only the first of these last two expression formats can be used if the *write(X)* subgoal is replaced by a *read(X)* subgoal.

But consider the following clause:

$p :- m, \text{assert}(n), n.$

Assume subgoals *m* and *n* are pure; clearly *assert* is not. Because *m* may be checking some set of preconditions for the *assert* subgoal, *m* must execute successfully before the *assert* subgoal can execute, as above. But notice that the *assert* subgoal can affect the computation of the following pure subgoal *n* since it modifies the definition of *n* [Shapiro 86]. In this example then, the pure subgoal following the side-effect subgoal may execute only after the preceding side-effect subgoal has successfully completed. Although it is shown in Section 13.9 how to relax this constraint, for now the following rule is adopted:

Sequencing Rule:

All subgoals preceding a side-effect subgoal must complete successfully before the side-effect subgoal can execute. Further, all subgoals following a side-effect subgoal must not begin execution until the preceding side-effect subgoal has successfully completed.

Figure 13.5 illustrates this rule. Note that the leaves of the execution tree can be divided into alternating segments: pure, then a side-effect subgoal, then pure, then a side-effect subgoal, and so on. These segments must execute in order, but within a pure segment, the subgoals can execute in any order, and possibly in parallel. Because and-parallelism is used within these segments, care must be taken to ensure that no two subgoals execute in parallel if they share a common, unbound variable; but this is easily accomplished with the normal RAP execution graph expressions.

-- figure 13.5 here --

13.7 Compiling Side-Effect Expressions

Given a clause to compile, if the clause is pure and thus contains only pure subgoals, these subgoals may execute in parallel, in any order, within the limits imposed by the normal execution graph expressions. In other words, the subgoals can execute in parallel except when they are not independent. The RAP graph expressions specify these conditions. But suppose one of the subgoals is a side-effect built-in. In particular, consider the clause:

$s2 :- a, s1, b.$

where subgoals a and b are pure and $s1$ is a side-effect built-in. (For simplicity, assume also that these subgoals have no arguments and thus are independent.) Clearly, these three subgoals must execute sequentially (actually, as discussed above, we consider below cases where b can execute before $s1$, or even before a). Because these subgoals must execute sequentially, it is tempting to compile this clause into the graph-expression:

$s2 :- (SEQ\ a\ s1\ b).$

And indeed, this is correct to a point. But note that because $s1$ is a side-effect built-in, the $s2$ clause becomes a side-effect clause, and thus $s2$ becomes a side-effect procedure.

Now consider the clause:

$s3 :- c, s2, d.$

where c and d are again pure but $s2$, from above, is a (potential) side-effect goal. Clearly then, the $s3$ clause is also a side-effect clause and $s3$ becomes a side-effect procedure. If we compile this clause in the same manner, yielding:

$s3 :- (SEQ\ c\ s2\ d).$

and if we then invoke the goal:

?- $s3.$

all subgoals in the two clauses will execute sequentially and not in parallel.

Note that the resulting execution tree is that shown in Figure 13.6. The left-to-right order of the leaves of the tree defines the normal execution sequence that is induced by a sequential Prolog. Note also that there are two pure subgoals preceding the single side-effect, leaf subgoal, and there are two following it. The two preceding it should be able to execute in parallel as should the two that follow it, as they are clearly independent. The compiled clauses above do not allow this, but they are certainly correct and do provide a proper sequencing of side-effects. But they unnecessarily limit the amount of parallelism. To correct this situation, the graph expressions need to be augmented with special synchronizing mechanisms. These mechanisms and their use are described below.

-- figure 13.6 here --

13.7.1 Synchronization Blocks

Simple two-part memory data structures are all that is required to implement the synchronization between the disjoint, parallel executing parts. These structures are called simply enough *synchronization blocks*, or *synch-blocks*. They can be implemented as two-word memory blocks, as shown in Figure 13.7. The first word is used to maintain a count of expressions involved in the synchronization, while the second word is used for signalling the

completion of a side-effect predicate.⁴

----Figure 13.7 here----

Five operations are defined for manipulating these synchronization blocks: create, inc, dec, signal, and wait. Each of these is defined below.

First, *create* simply creates a synch-block in free memory. Two types of creation are allowed, one for creating input synch-blocks and one for creating output synch-blocks. The created synch-blocks are identical in both cases, but the fields are initialized to different values depending on which type of creation is invoked. The use of these two types of synch-blocks is discussed below.

create(sb,input) - creates a synch-block for the left part of the and-execution tree preceding a side-effect predicate;

sb.ecnt := 0 {indicates no unfinished, preceding, pure expressions}
sb.signal := yes {indicates no unfinished, preceding,
side-effect built-in}

create(sb,output) - creates a synch-block for the right part of the and-execution tree following a side-effect predicate;

sb.ecnt := 1 {indicates one unfinished, preceding side-effect built-in}
sb.signal := no {indicates that the preceding side-effect
built-in has not completed}

The expression-count (*ecnt*) field of a synch-block can be either incremented or decremented atomically. The statement **inc(sb)** is shorthand for

sb.ecnt := sb.ecnt + 1;

where the addition must be performed atomically. Similarly, the statement **dec(sb)** is shorthand for the statement

sb.ecnt := sb.ecnt - 1;

where the subtraction must also be performed atomically.

A signal field indicates whether the associated side-effect built-in has yet completed. A value of "no" indicates that the predicate has not yet completed, while a value of "yes" indicates that it has completed. A signal field may be changed to "yes" with the statement **signal(sb=yes)**.

A **wait(sb.ecnt=0)** statement indicates that the expression-count field of the designated synch-block is to be checked for a value of zero. If the value is zero, the processor continues execution of the current expression. If the value is non-zero, the expression evaluation is suspended and the expression is placed in a suspended expression list. The manner in which suspended expressions are reawakened is discussed in Section 13.10.

Similarly, a **wait(sb.signal=yes)** tests the signal field of a synch-block for a value of "yes". If such a value is found, execution continues. Otherwise, execution is suspended, and the expression is placed in the suspended expression list.

⁴ Actually, if not for soft side-effects, a single word synch-block could be used.

13.7.2 Distributing Synchronization Blocks

Consider now a list of subgoals which are all pure except for a single subgoal in the middle of the list which has a side-effect. As described above, the list must now be broken into three segments: 1) the pure subgoals preceding the side-effect subgoal, 2) the side-effect subgoal itself, and 3) the pure subgoals following the side-effect subgoal. These three segments must execute in order. When one segment finishes, it must somehow "signal" the following segment that it has completed. Similarly, each segment must wait for the preceding segment to signal it. Figure 13.8a illustrates the use of synch-blocks between segments to provide a correct sequencing of subgoals.

Clearly the synch-blocks must be properly distributed to the appropriate segments. To do this, the RAP execution graph expressions must be extended to carry along the required synch-blocks and pass them down the and-execution tree in order to reach the appropriate segments. If not done properly, this can add significant additional overhead. In the technique discussed below, it is shown how this overhead is minimized.

Before considering the extensions to the RAP expressions, consider a single pure subgoal. It must 1) wait on the preceding side-effect goal to complete, and 2) it must signal the following side-effect goal when it itself completes. Call the preceding side-effect goal s_1 and the following side-effect goal s_2 . The pure subgoal under consideration lies between s_1 and s_2 . A single synch-block can be used by this pure subgoal to effect the required synchronization. The signal field of the synch-block will be used by the preceding side-effect goal, s_1 , to signal its completion, and the ecnt field can be used by the pure subgoal to indicate to the following side-effect subgoal, s_2 , when it has completed. This process is illustrated in Figure 13.8a.

-- figure 13.8 here --

Generally, there are more than one pure subgoal between any two side-effect goals. They can all utilize a common synch-block, however, to synchronize their executions with the side-effect goals. Figure 13.8b illustrates this. It can be seen in this figure that the signal field can be a simple boolean value field, while the ecnt field must be an integer. Given this simple scheme, the number of side-effect goals can be increased; to do so, all that is required is to add one more synch-block between each pair of side-effect goals and to make this synch-block accessible to all intervening pure subgoals, as shown in Figure 13.9. In this figure, it can be seen that a side-effect goal needs access to two synch-blocks - the preceding one and the following one. The preceding synch-block is called an *input* synch-block; the following synch-block is called an *output* synch-block. Pure goals need access to only one synch-block; this synch-block is called an *input* synch-block, although in fact it serves both an input and an output function (see the figures).

-- figure 13.9 here --

Now the extensions to the RAP expressions can be considered. These extended expressions represent code macros that are expanded in-line before actual code-generation. In

these expressions, *seg* stands for "side-effect goal", and *seb* stands for "side-effect built-in".

1. (pure(SI) *e*)
where *e* is a pure expression, gets expanded into the sequential code sequence:
 wait(SI.signal = yes)
 e
 dec(SI)
2. (seb(SI,SO) *s*)
where *s* is a side-effect built-in or sequence of side-effect built-ins, gets expanded into the sequential code sequence:
 wait(SI.ecnt = 0)
 s
 signal(SO.signal=yes)
 dec(SO)
If *s* is a sequence of side-effect built-ins, they execute sequentially.
3. (seg(SI,SO) *p*<args>)
where *p* is a potential side-effect goal but is not a side-effect built-in, gets expanded into:
 call *p*(<args>,SI,SO)
4. *f*(<args>,SI,SO) :- <graph expression for side-effect clause body>
If *f* is a side-effect procedure and this is a side-effect clause, then the clause head is extended to accept the two additional synchronization block parameters. All calls to *f* will have been extended to pass these two synch-blocks to *f*.
5. *f*(<args>,SI,SO) :- (pure(SI) *e*)
where *f* is a side-effect procedure but this clause is a pure-clause within *f*. Since this clause is pure, the clause body is pure, and a normal, simple RAP graph expression is compiled for the body. Because *f* is a side-effect procedure, all calls to *f* will be side-effect goals, and so they will have been augmented with the two extra synch-block parameters. The clause head must therefore be extended to accept these two extra parameters, even though only the SI synch-block is used. (Actually, an optimizing compiler can simply ignore the SO synch-block. It is included in the source code simply to maintain consistency between the use and declaration of a procedure.) Note, however, from the definition of a pure expression that neither synch-block is passed to any subgoal in the body expression. This is an efficiency advantage.
6. All pure procedures are compiled normally, without any induced overhead whatsoever.

These additional expressions, along with the statements for manipulating synch-blocks, are sufficient to ensure the correct sequencing of pure code segments and side-effect goals.

Note that all non-pure expressions receive either one or two synch-blocks. One is generally an input synch-block, denoted as SI, and one is an output synch-block, denoted as

SO. An input synch-block represents the preceding side-effect goal and the number of preceding pure expressions that follow the preceding side-effect goal. The synch-block's two components — *ecnt* and *signal* — represent these two pieces of information. Since the synch-block must be distributed to the preceding side-effect goal, to each preceding pure expression, and to the consuming side-effect goal, the *ecnt* field is used to count the number of times the synch-block has been distributed. Each time the synch-block is distributed, the *ecnt* field is incremented. When a pure expression or the preceding side-effect goal successfully completes, it decrements the *ecnt* field. When the *ecnt* field reaches zero, all preceding pure expressions and the preceding side-effect goal have completed. At this point, the next side-effect goal can begin execution. When a side-effect goal completes, it sets the *signal* field of its output synch-block to "yes", indicating that it has completed. At this point, the following pure expressions which precede the next side-effect goal can all begin execution.

13.8 Example Side-Effect Graph Expressions

To illustrate the handling of side-effect goals in the RAP model, this section presents several examples of side-effect clauses and their corresponding side-effect graph expressions.

Example 1

First, consider the the following clause and query:

```
f :- a, s, b.  
?- f.
```

where *a* and *b* are pure but *s* is a side-effect built-in. Because *s* is a side-effect built-in, the clause for *f* is a side-effect clause, and thus *f* is a side-effect procedure. The clause is therefore compiled as:

```
f(SI,SO) :-  
  (SEQ inc(SI)  
    inc(SO)  
    (PAR (pure(SI) a)  
        (seb(SI,SO) s)  
        (pure(SO) b))).
```

After macro expansion, this clause becomes:

```
f(SI,SO) :-  
  (SEQ inc(SI.ecnt)  
    inc(SO.ecnt)  
    (PAR (pure(SI) a)  
        wait(SI.signal = yes)  
        a  
        dec(SI.ecnt)  
        (seb(SI,SO) s)
```

```
wait(SI.ecnt = 0)
s
signal(SO.signal = yes)
dec(SO.ecnt)
(pure(SO) b)
wait(SO.signal = yes)
b
dec(SO.ecnt)))
```

Because f is a side-effect procedure, the query,
?- f .
is compiled into the sequential code:

```
(SEQ create(SI,input)
      SI.ecnt := 0
      SI.signal := yes
      create(SO,output)
      SO.ecnt := 1
      SO.signal := no
      call f(SI,SO)).
```

When this query expression is executed, two synch-blocks are created and initialized. The SI signal field is initialized to "yes" as there is no unfinished, preceding side-effect goal, and the SI ecnt field is initialized to 0 since there are no unfinished, preceding pure expressions. The SO synch-block represents the side-effect goal and the following pure subgoals that precede the next side-effect subgoal, if any (here, there are none). The SO signal field is initialized to "no" since the side-effect goal s has not yet completed; the ecnt field is set to 1, representing the unfinished, preceding side-effect goal, namely f .

Following creation and initialization of the synch-blocks, the subgoal call is executed, and the procedure for f is entered. Upon entry, both synch-blocks have their ecnt fields incremented. This is because the clause has both a left and a right pure part, and because the corresponding synch-blocks will be passed into these expressions. These expressions must be counted before the expressions are entered and the synch-blocks passed in. This explains why the creation and incrementing occur sequentially before the parallel subgoal expressions begin.

Then the PAR expression is entered. Upon entry, the SI synch-block has SI.ecnt=1 and SI.signal=yes, while the SO synch-block has SO.ecnt=2 and SO.signal=no. Because a PAR expression was entered, all three subexpressions can begin executing in parallel. The second expression, the *seb* expression, suspends since SI.ecnt \neq 0, and the third expression suspends since SO.signal \neq yes. Only the first expression can execute since SI.signal=yes. Thus subgoal a begins execution. When and if a successfully completes, SI.ecnt is decremented to 0. This is the condition the second expression is awaiting. When SI.ecnt is found to be zero, the side-effect subgoal s begins execution. When and if it successfully completes, SO.signal is set to yes, and SO.ecnt is decremented to 1. When SO.signal is found to be yes, the third subgoal, b , can begin execution. Upon successful completion of b , SO.ecnt is decremented to 0. Since all three subexpressions completed successfully, the entire expression has completed, and control returns to the top-level query processor. Both synch-blocks have their signal fields set to yes and their ecnt fields set to zero, indicating completion of all subgoals. (It should be obvious

that the PAR expression can be replaced here by the more efficient SEQ expression. As described in Section 13.9 however, soft-side effects benefit from the PAR expression. In order to hopefully minimize confusion, only PAR expressions are used throughout the remainder of this chapter.)

Note that subgoal *a* may invoke an arbitrarily complex subtree of subgoals, one perhaps containing several thousands of nested subgoals. But these subgoals are all pure and so are compiled into normal execution graph expressions. These expressions do not have to consider any synch-blocks, and no synch-blocks are passed in as parameters. Thus no overhead is introduced into this subtree for the required synchronization. The same is true for the third subgoal, *b*. In fact, the only operations on the synch-blocks are those shown explicitly in the compiled expression above. It can be seen then that the root node of a pure subtree in an and-execution tree handles the synch-block manipulation for the entire pure subtree below it. No overhead is introduced into the subtree except at the root node, and this overhead is minimal. This low overhead of passing and manipulating synch-blocks is a significant advantage of the RAP model.

If you consider the resulting macro-expanded code above, it can be seen that an optimizing compiler can easily make improvements. For example, reconsider the complete macro-expanded code for the clause:

```
f :- a, s, b.
```

It is:

```
f(SI,SO) :-  
  (SEQ inc(SI.ecnt)  
    inc(SO.ecnt)  
    (PAR (pure(SI) a  
          wait(SI.signal = yes)  
          a  
          dec(SI.ecnt)  
        (seb(SI,SO) s  
          wait(SI.ecnt = 0)  
          s  
          signal(SO.signal = yes)  
          dec(SO.ecnt)  
        (pure(SO) b  
          wait(SO.signal = yes)  
          b  
          dec(SO.ecnt))))
```

Careful consideration of this code shows that certain instructions can be omitted due to the fact that *s* is a side-effect built-in and that the PAR expression can be replaced by a SEQ expression. In the code below, the highlighted parts are the required parts; the remainder can be optimized away.

```
f(SI,SO) :-  
  (SEQ inc(SI.ecnt)  
    inc(SO.ecnt)  
    (PAR (pure(SI) a  
          wait(SI.signal = yes)
```

```

a
  dec(SI.ecnt)
(seb(SI,SO) s)
  wait(SI.ecnt = 0)
s
  signal(SO.signal = yes)
  dec(SO.ecnt)
(pure(SO) b)
  wait(SO.signal = yes)
b
  dec(SO.ecnt)))
```

Extracting only the above highlighted parts, the resulting, optimized code is simply:

```
f(SI,SO) :-
  (SEQ wait(SI.signal = yes)
    a
    wait(SI.ecnt = 0)
    s
    signal(SO.signal = yes)
    b
    dec(SO.ecnt))
```

The resulting, optimized code is clearly more efficient. However, since the unoptimized code is clearly correct as well, and since it is easier to represent, the unoptimized code sequences are used throughout the remainder of this paper. Additional optimization techniques based on data dependency analysis are discussed in [DeGroot 88a].

Example 2

Now consider the following query and two clauses:

```
?- f.
f :- a, s1, b.
s1 :- c, s2, d.
```

where subgoals *a*, *b*, *c*, and *d* are pure, and *s2* is a side-effect built-in. Then *s1* is a side-effect procedure, making *f* a side-effect procedure as well. As before, the query is compiled into:

```
(SEQ create(SI,input)
  create(SO,output)
  call f(SI,SO)).
```

The two clauses are compiled nearly identically as:

```
f(SI,SO) :-  
  (SEQ inc(SI)  
   inc(SO)  
   (PAR (pure(SI) a)  
        (seg(SI,SO) s1)  
        (pure(SO) b))).5
```

```
s1(SI,SO) :-  
  (SEQ inc(SI)  
   inc(SO)  
   (PAR (pure(SI) c)  
        (seb(SI,SO) s2)  
        (pure(SO) d))).
```

This example is possibly more interesting. Once again, when the query is activated, both an input and an output synch-block are created. Then the side-effect procedure f is called with these two synch-blocks, and the clause for f is entered. Upon entry, $SI.ecnt$ is incremented to 1, counting the preceding pure expression for a , and $SO.ecnt$ is incremented to 2, counting the following pure expression for b and the nested side-effect built-in. The PAR expression is then entered.

The third subexpression, for b , cannot execute since $SO.signal \neq yes$. But the first two subexpressions can begin parallel execution. Suppose they do. The first subexpression invokes a pure execution subtree for subgoal a which begins immediate execution and does so without any overhead of the synchronization mechanism. The second subexpression begins parallel execution and immediately calls the side-effect procedure $s1$, passing in the modified input and output synch-blocks, SI and SO . This call results in the clause for $s1$ being executed.

When this clause begins execution, both synch-blocks are again incremented, thereby counting the left and right pure subexpressions in the clause. Now, $SI.ecnt=2$ and $SO.ecnt=3$, and $SI.signal=yes$ and $SO.signal=no$. Consequently, when the PAR expression is entered, only the first subexpression can begin execution. The second and third subexpressions must suspend. When the first subexpression, for c , begins execution, c may also invoke an arbitrarily complex execution subtree. Note that this subtree can execute in parallel with the subtree invoked by subgoal a .

Before the side-effect subgoal $s2$ can execute, both subgoals a and b must complete. If the execution subtrees for a and b are executed successfully, control returns to the pure subexpressions containing them. These subexpressions then proceed to decrement the $SI.ecnt$ field. Since $SI.ecnt=2$ at this time, the two decrements set $SI.ecnt=0$. Because both subexpressions may be executing in parallel and may possibly attempt to simultaneously decrement $SI.ecnt$, the decrement operation must be atomic, using semaphores, test-and-set instructions, replace-add instructions, or others. Eventually, however, $SI.ecnt=0$.

At that point, the second subexpression of the $s1$ clause can begin execution. This subexpression, with its macro expansion, is:

⁵As just shown, the code for this clause can actually be significantly optimized.

```
(seb(SI,SO) s2)
  wait(SI.ecnt = 0)
  s2
  signal(SO.signal = yes)
  dec(SO)
```

After the wait instruction succeeds, the side-effect built-in *s2* can execute. If successful, *SO.signal* is set to *yes* to indicate its completion to all following pure subgoal expressions, and *SO.ecnt* is decremented to 2.

When *SO.signal* is set to *yes*, the two subexpressions

```
(pure(SO) b)))
  wait(SO.signal = yes)
  b
  dec(SO.ecnt)
```

in the clause for *f*, and

```
(pure(SO) d)))
  wait(SO.signal = yes)
  b
  dec(SO.ecnt)
```

in the clause for *s1*, can both begin execution. This allows the two subgoals *b* and *d* to invoke their pure execution subtrees in parallel, and again without any overhead of the synchronization mechanism. Figure 13.10 illustrates the resulting parallel execution. Compare it with the sequential execution of Figure 13.7, and compare the above execution expressions with the expressions in Section 13.7.

This example shows how, in a sequence of nested clauses, pure subgoals preceding a side-effect goal are all allowed to execute in parallel, then the side-effect built-in executes, and then the following pure subgoals are allowed to execute in parallel. Obviously, this example extends to nested clauses of any depth.

-- figure 13.10 here --

Example 3

As mentioned before, a side-effect procedure can contain both side-effect clauses and pure clauses. When a pure clause is contained within a side-effect procedure, it is compiled simply as follows:

```
f(SI,SO) :-
  (pure(SI) e).
  wait(SI.signal = yes)
  e
  dec(SI.ecnt)
```

Example 4

Consider the following example clauses and their compilations. In each clause, let a , b , and c be pure subgoals, let $s1$ and $s2$ be side-effect built-ins, and let $g1$ and $g2$ be non-built-in, side-effect subgoals. The clause:

$f :- a, s1, s2, b.$

is compiled as:

```
f(SI,SO) :-  
    inc(SI.ecnt)  
    inc(SO.ecnt)  
    (PAR (pure(SI)  a)  
         (seb(SI,SO) s1 s2)  
         (pure(SO)  b)).
```

Here, the two side-effect built-ins execute sequentially as required.

The clause:

$f :- a, g1, b, g2, c.$

is compiled into:

```
f(SI,SO) :-  
    create(SM,output)  
    inc(SI.ecnt)  
    inc(SM.ecnt)  
    inc(SO.ecnt)  
    (PAR (pure(SI)  a)  
         (seg(SI,SM) g1)  
         (pure(SM)  b)  
         (seg(SM,SO)  g2)  
         (pure(SO)  c)).
```

This example shows that some synch-blocks are created within a clause and not just at the query level. An optimizing compiler can combine the creation of SM with the incrementing of SM.ecnt.

Example 5

As discussed above, a side-effect clause that contains only a side-effect built-in can be compiled with a SEQ expression rather than with a PAR expression. The resulting code can even be highly optimized. When the side-effect goal is not a built-in however, the PAR expression must be used in order to extract greater parallelism. Consider a complex clause containing both side-effect goals and side-effect built-ins:

$f :- a, g1, b, s1, c, g2, d.$

where as before, a , b , c , and d are pure subgoals, $g1$ and $g2$ are side-effect subgoals (but not built-ins), and $s1$ is a side-effect built-in. This clause can be compiled as:

```
f(SI,SO) :-
```

```
create(SM1,output)
create(SM2,output)
inc(SI.ecnt)
inc(SO.ecnt)
inc(SM1.ecnt)
inc(SM2.ecnt)
(SEQ (PAR (pure(SI) a)
          (seg(SI,SM1) g1)
          (pure(SM1) b))
      (seb(SM1,SM2) s1)
      (PAR (pure(SM2) c)
           (seg(SM1,SO) g2)
           (pure(SO) d))).
```

As above, significant code optimizations can be performed by an optimizing compiler. The entire clause body can instead be wrapped in a single PAR expression.

Example 6

Two examples are now shown for clauses that do not have pure parts preceding or following a side-effect goal. First, the clause

```
f :- a, s1.
```

is compiled as:

```
f(SI,SO) :-
  inc(SI)
  (PAR (pure(SI) a)
       (seg(SI,SO) s1)).
```

The clause

```
f :- s1, b.
```

is compiled as:

```
f(SI,SO) :-
  inc(SO)
  (PAR (seg(SI,SO) s1)
       (pure(SO) b)).
```

Example 7

Finally, consider an example in which both the side-effect expressions and independence-determination expressions are combined. Consider the clause:

```
f(X) :- a(X), s(X), b(X), c(X).
```

Let a , b , and c be pure subgoals and s be a non-built-in, side-effect subgoal. This clause is compiled as:

```
f(X,SI,SO) :-
  (SEQ inc(SI)
   inc(SO)
   (GPAR(X) (pure(SI) a(X))
            (GPAR(X) (seg(SI,SO) s(X))
                     (pure(SO) (GPAR(X) b(X) c(X))) )))
```

Here, if f is entered with X ground, all four subgoals will begin parallel execution. The additional side-effect synchronization code, however, will ensure that the four subgoals, including all their descendant subgoals, execute in the proper sequence. If X is not ground on entry, a executes first, sequentially. Subgoals s , b , and c , must wait. In this case, s cannot be allowed to begin expansion into its subtree, for any pure subgoals in the leftmost part of its tree must now wait for a 's subtree to complete and hopefully present a ground value for X . It can be seen then that the normal RAP execution graph expressions provide data synchronization, while the synch-block-management code extensions provide side-effect synchronization.

13.9 Soft Side-Effect Built-Ins

It is possible to divide the set of side-effect built-ins into *soft side-effects*, those that cannot affect the following computation, and *hard side-effects*, those that can. Certainly, a *write* subgoal cannot affect the execution of the following pure subgoals and so it is a soft side-effect goal. An *assert* subgoal, however, may very well affect the following execution. Given these definitions, the *seb* expression defined above is for hard side-effect subgoals. A soft side-effect built-in s might be compiled as:

```
(soft-seb(SI,SO) s)
  signal(SO.signal=yes)
  wait(SI.ecnt = 0)
  s
  dec(SO)
```

Here, because the signal operation occurs before the side-effect subgoal s begins execution, the pure subgoals following s can immediately begin execution. The side-effect goals still execute in sequential order, however, as required, and still, no-side-effect goal can execute until all preceding goals have completed.

But there is no sense in waiting until just before s begins execution to set the signal field to yes; it can instead be set at the creation of the synch-block. To see how, note that once side-effect built-ins are classified as either hard or soft, the same classification applies to clauses and procedures. Then we need to be able to create both hard and soft synch-blocks. A new `create_soft` operation is defined:

```
create_soft(sb,output) - creates an output synch block for a soft side-effect expression
  sb.ecnt := 1; {indicates to the following side-effect goal that there
                 is at least one preceding, incomplete side-effect goal,
                 namely, the soft one for which this synch-block is
                 being created}
```

```
sb.signal := yes; {indicates to all following pure expressions that
                    they do not have to wait on this soft-side effect
                    goal}
```

The corresponding definition of **soft-seb** is then:

```
(soft-seb(SI,SO) s)
  wait(SI.ecnt = 0)
  s
  dec(SO)
```

The definition of **soft-seg** remains the same as **seg**. A clause containing both hard and soft side-effects is considered to have only hard side-effects. Now consider the following example:

```
f :- a, s1, b.
s1 :- d, write(foo), e.
?- f.
```

Let *a*, *b*, *d*, and *e* be pure subgoals. Clearly, *write(foo)* is a soft side-effect subgoal, and thus *s1* and *f* are a soft side-effect procedures. The query is compiled as:

```
?- create(SI,input)
    SI.ecnt := 0;
    SI.signal := yes;
    create-soft(SO,output)
    SO.ecnt := 1;
    SO.signal := yes;
    call f(SI,SO).
```

The two clauses would be compiled as:

```
f :- a, s1, b.
f(SI,SO) :-
  (SEQ inc(SI)
    inc(SO)
    (PAR (pure(SI) a)
      (soft-seg(SI,SO) s1)
      (pure(SO) b))).

s1 :- d, write(foo), e.
s1(SI,SO) :-
  (SEQ inc(SI)
    inc(SO)
    (PAR (pure(SI) d)
      (soft-seb(SI,SO) write(foo))
      (pure(SO) e))).
```

This will allow all of the pure subgoals a , b , d , and e to execute in parallel. However, the *write* subgoal can execute only when both a and b have successfully completed, as required.

Soft side-effect goals are those that cannot affect the computation of their following pure subgoals; as a consequence, the following pure subgoals do not have to wait for the preceding soft side-effect goal to complete. However, two or more soft side-effect subgoals must still maintain their proper sequence. For example, every read goal can be properly sequenced with the normal execution graph expressions, as demonstrated above. However, two read subgoals must not be allowed to execute out of order. The soft side-effect control constructs ensure that this does not occur.

As explained above, if a procedure contains both hard and soft side-effect subgoals, all are treated as hard side-effects. This may possibly be improved. Other optimizations are possible and constitute future research efforts. For example, two reads from two different files do not need to be sequenced. Also, reads followed by writes may not need to be sequenced with synch-blocks, as the normal graph expressions will likely ensure correct sequencing.

13.10 Suspending Expression Execution

When a wait instruction is executed in an expression, a particular condition is checked. If the condition exists, execution of the expression may continue. If the condition does not exist, the expression execution must suspend. Later, when and if the condition becomes true, the suspended expressions may resume execution.

Clearly there must be an execution mechanism capable of providing this service. Many are possible. One possible mechanism is a simple *suspended expression list*. When an expression checks for a condition and finds it false, the expression is placed on the suspended expression list, and the processor looks for another expression to execute. As long as the processor has active expressions to execute, it will not be idle, and system utilization will not suffer. If a condition required by a suspended expression becomes true while the processor is executing other expressions, the condition does not need to be noted by the waiting processor. But when a processor runs out of active expressions to execute, it can then make a pass through the suspended expression list, rechecking the required conditions for each suspended expression. As soon as one condition is found to be true, execution of the associated expression can resume. If no condition is found to be true, the processor has run out of expressions to execute, and it can then volunteer to assist other processors in executing their overload of active expressions, if any. Although this model is a polling model, it will not flood the communication system with condition-checking memory accesses except when a processor is idle and no other processor will send it additional work to do. But in such cases, the overall system activity will most likely be low, and the communication network will thus likely be underutilized. Other architectural implications of the required support mechanisms can be found in [Hermenegildo 87].

13.11 Summary

An and-parallel execution model for Prolog has been presented; this model is called the Restricted And-Parallelism (RAP) model. A technique for extending the RAP model to handle side-effect computations has also been described. This technique ensures that the normal,

observable sequence of side-effects that would occur in a traditional, sequential execution of a Prolog program is retained within the restricted and-parallel execution of the same program. To handle side-effects within the RAP model, a new set of expressions were introduced into the execution graph expression language. These additional expressions are sufficient to ensure proper sequencing of side effects.

The only data structures required are two-word synch-blocks. The extended execution expressions operate on these synch-blocks to properly serialize the side-effect goals and the intervening pure subgoals. The model described actually results in little additional overhead, as the synch-blocks are distributed down into the and-tree only as far as required. Possible compiler optimizations were presented to further reduce the overhead, and side-effect goals were divided into hard side-effects and soft side-effects.

It should be pointed out that the model is not yet complete. At present, the backtracking semantics and an efficient implementation model are being investigated. An automatic graph expression compiler is also being extended to incorporate the side-effect expressions [DeGroot 88a].

Bibliography

- [Carlton 88] Mike Carlton and Peter Van Roy, "A Distributed Prolog System with And-Parallelism," Dept. of EECS, Univ. of California at Berkeley, submitted to the 1988 Hawaii International Conference on System Sciences, 1988.
- [Chang 85] Jung-Herng Chang, Alvin Despain, and Doug DeGroot, "AND-Parallelism of Logic Programs Based on a Static Data-Dependency Analysis," *Procs of the Spring Compcon 85*. IEEE Computer Society Press, 1985, pp. 218-225.
- [Clark 86] Keith Clark and Steve Gregory, "PARLOG: Parallel programming in logic," *ACM Transactions on Programming Languages and Systems*, January, 1986, pp. 1-49.
- [Conery 81] John Conery and Dennis Kibler, "Parallel Interpretation of Logic Programs," *Procs. of the Conf. on Functional Programming Languages and Computer Architecture*, ACM, 1981, pp. 163-170.
- [Conery 87] John Conery, *Parallel Execution of Logic Programs*, Kluwer Academic Publishers, 1987.
- [Crammond 85] Jim Crammond, "A comparative study of unification algorithms for or-parallel execution of logic languages," *Procs. of the 1985 Int'l Conf. on Parallel Processing*, Doug DeGroot, Editor, IEEE Computer Society Press, 1985, pp. 131-138.
- [DeGroot 84] Doug DeGroot, "Restricted And-Parallelism," *Proceedings of the International Conference on Fifth Generation Computer Systems 1984*, North Holland, 1984, pp. 471-478.
- [DeGroot 85] Doug DeGroot and Jung-Herng Chang, "A Comparison of Two And-Parallel Execution Models," *Hardware and Software Components and Architectures for the 5th Generation*, AFCET Informatique, March 1985, Paris, pp. 271-280.
- [DeGroot 87a] Doug DeGroot, "Restricted And-Parallelism and Side-Effects," *Procs. of the Symposium on Logic Programming*, IEEE Computer Society, San Francisco, 1987.
- [DeGroot 88a] Doug DeGroot, "A Technique for Compiling Execution Graph Expressions for Restricted And-Parallelism " *Proceedings of the 1987 International Supercomputing Conference*, Athens, June 8-12, 1987, Dave Kuck and Constantine Polychronopolous, Editors, Springer Verlag, to be published, 1988.
- [DeGroot 88b] Doug DeGroot, "And-Parallelism in Logic Programs," in *Advanced Semiconductor Technology and Computer Systems*, Guy Rabbat, Ed., Van Nostrand Reinhold, New York, to be published, 1988.

- [Halstead 88] Robert Halstead, "Design Requirements of Concurrent Lisp Machines," *Supercomputers and AI Machines*, Kai Hwang and Doug DeGroot, Eds., Chapter 3, McGraw-Hill, 1988, (this volume).
- [Hermenegildo 87] Manuel Hermenegildo, *A Restricted And-Parallel Execution Model and Abstract Machine for Prolog Programs*, Kluwer Academic Press, 1987.
- [Kluzniak 84] Feliks Kluzniak and Stanislaw Szpakowicz, *Prolog for Programmers*, Academic Press, 1984.
- [Lloyd 84] John Lloyd, *Foundations of Logic Programming*, Springer-Verlag, New York, 1984.
- [Mellish 81] Chris Mellish, "The Automatic Generation of Mode Declarations for Prolog Programs," DAI Research Paper 163, Dept. of Artificial Intelligence, Univ. of Edinburgh, August 1981.
- [O'Keefe 87] Richard O'Keefe, A Prolog program to detect side-effect procedures, personal correspondence, Quintus Computer Systems, Mountain View, California, May, 1987.
- [Pollard 81] G. H. Pollard, *Parallel Execution of Horn Clause Programs*, Ph.D. dissertation, University of London, Imperial College of Science & Technology, United Kingdom, 1981.
- [Shapiro 83] Ehud Shapiro, "A Subset of Concurrent Prolog and Its Interpreter," ICOT Tech. Report TR-003, ICOT, Tokyo, February, 1983.
- [Shapiro 86] Ehud Shapiro, *The Art of Prolog*, MIT Press, 1986.
- [Ueda 87] Kazunori Ueda, *Guarded Horn Clauses*, MIT Press, Cambridge, 1987.
- [Warren 80] David H. D. Warren, "Efficient Processing of Interactive Relational Database Queries Expressed in Logic," *Procs. of the 7'th International Conference on Very Large Data Bases*, July, 1980, pp. 181-203.
- [Warren 83] David H. D. Warren, "An Abstract Prolog Instruction Set," Tech. Note 309, SRI International, Oct. 1983.
- [Wise 86] Michael Wise, *Prolog Multiprocessors*, Prentice/Hall International editions, 1986.