UNIVERSIDAD POLITÉCNICA DE MADRID
FACULTAD DE INFORMÁTICA

Some Contributions to the Study of Parallelism and Concurrency in Logic Programming

PhD. Thesis

MANUEL CARRO LIÑARES

April, 2001

PhD. Thesis

Some Contributions to the Study of Parallelism and Concurrency in Logic Programming

presented at the Computer Science School
of the Technical University of Madrid
in partial fulfillment the requirements for the degree of
Doctor in Computer Science

PhD. Candidate:	Manuel Carro Liñares Licenciado en Informática Universidad Politécnica de Madrid
Advisor:	Manuel V. Hermenegildo Salinas Catedrático de Universidad

Madrid, April, 2001

To my family

Acknowledgments

In the first place, I owe gratitude to my family, specially to my father and my mother (may her soul rest in peace), for waiting so much, for their always supporting me, and for understanding me more than I understood them.

I have a very special debt of gratitude to my PhD advisor for all he has taught me along all these years with neverending strength and stamina. My appreciation also to my laboratory fellows, among which I will mention Francisco Bueno, Daniel Cabeza, Maria Jose García de la Banda, Pedro López and Germán Puebla.

Some of the implementations of the ideas shown in the present piece of work have been made in collaboration with other people. My gratefulness to them, specially to Luis Gómez for his implementation of *VisAndOr*, to Angel López for his work in *APT*, to José Manuel Ramos for his work in *VIFID*, to Göran Smedbäck for his work in *TRIFID* and *ProVRML*, and to María José Fernández for her work in *IDRA*.

I should also mention others I have worked with and discussed different aspects of our work. I hope to have learned from them. Special thanks should go to Enrico Pontelli, Kish Shen, Gopal Gupta, DonXing Tang, Saumya Debray, Abder Aggoun, Herbert Simonis, Jonas Barklund, Hakan Millroth, Johan Bevemyr, Johan Montellius, V. Santos Costa, Khayri Ali. Those omitted by my forgetfulness, please accept my apologies.

I am also grateful to the Spanish government for the concession of a research scholarship and several research projects, to the ESPRIT program of the European Economic Union for several research projects, and to the Fulbright foundation, for a Spanish-US collaboration grant. All of them have made the accomplishment and presentation of the works which conform my thesis easier.

Last, but not least, a big **Thank You!** to all my friends who, without knowing it, supported and helped me.

Madrid, April the 27th, 2001

Summary

Declarative languages have some unique characteristics which make them invaluable in many applications involving a certain degree of *intelligence* or knowledge management. They also are, due to its high-level nature, less error-prone, and the programs coded using them are more amenable to being thoroughly checked using automatic tools aimed at uncovering flaws in their design or coding.

The use of such languages is, however, not as widespread as it could be expected from the above description. They are usually tagged as being inefficient, as lacking some important features, or as not having all the characteristics needed to interface with the *rest of the world* as needed by Computer Science today.

Focusing on the logic languages realm, we study some of the lacks commonly attributed to logic languages, propose some solutions, and evaluate their efficiency in real problems. More precisely, our work

- proposes visualization methods for sequential, parallel, and constraint logic languages;
- proposes some optimizations for the parallel execution of logic languages;
- proposes a method for evaluating the performance of the parallel execution of logic programs; and
- proposes a new method for expressing concurrency in logic languages.

All the points above have been evaluated in real systems.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Chapter Summary

An overview of the thesis is presented in this chapter. The internal relations of the work herein presented and how it stems from a sustained research line is clarified and put in perspective. We also show how the results obtained fit into the development of an evolving discipline in a world of changing technology. Last, and since new ideas and developments can seldom be solely attributed to a single individual, we clarify the contributions of the author and the work in every part of the thesis.

1.1 A Perspective of the Advances and Needs of Programming Languages

As the complexity of computer applications grows, new and more powerful tools and programming languages are needed. They, on one hand, should free the programmer from error-prone tasks and, on the other hand, should furnish her/him with advanced features, which would have been considered as exotic no long ago. Besides, the non-linearity of the process of software creation should not be forgotten: for reasons inherent to the human mind, loops and feedback from previous stages are frequent when developing a software project. One motivation of advanced programming languages is to minimize this looking backwards by making the path from design to implementation as straight and error-free as possible. Among the means to reach this goal we can obviously cite separating the programmer from low-level details, and using programming languages in which the distance from a conceptual solution, easy to understand solution, to the final code is as short as possible.

In any case, and with almost total certainty, the revision cycle will be used during long time in software engineering. Well designed, high level languages, are not enough: there is a need for good tools which allow understanding why a particular program behaves as it does. These tools should pair up with the abstraction level of the language and the applications to generate. As an example, it is not reasonable to study the behavior of a complex search procedure (say, a *Tabu* search) by tracing the assembler code produced by a compiler. Disregarding the possible compiler bugs, the behavior of the low level program will surely reflect what the higher-level code expresses.

There are many applications which need a complex internal technology, and which demand not only a nontrivial conceptual design, but also an important technological support. They are not just academic experiments: today's computer applications require very often solutions to difficult problems. We will mention some (non mutually exclusive) general cases of such applications, instances of which are not difficult to find in many computer programs used almost routinely.

- All the applications aimed at processing symbolic information. While information can be represented and managed numerically, the closer the representation is to the concepts to be coded, the easier programming the process to be performed is. Many applications sought during the first years of A.I. would fall into this category, including general problem solvers, planners, truth maintenance systems, knowledge representation methods, theorem provers, language processing... All of them are clear candidates to be expressed using a language with symbolic processing capabilities.[1]

- In many cases, applications have a strong search or automatic reasoning component which, as last resort and apart from knowledge representation issues, boils down to exploring a search space: verifying a theorem can, in a very simplistic formulation, be conceptually seen as a search for the right demonstration in the whole space of demonstrations, and the same can be applied to the elaboration of a plan. Some languages (e.g., Prolog, those providing finite domains, Oz, and others) provide builtin search procedures, which are often enough for many cases. However, in more complicated cases, the search has to be driven by the particular problem data and by the knowledge on the domain. Many times this driving is not straightforward, and a good, easy-to-access representation of such information helps in coding such a search. Advanced representations for data structures makes writing such customized search procedures easier—as with any other algorithms.

- In a next complexity class we find applications which, besides the above mentioned capabilities, need an architecture conceptually based on the interaction of

[1] This is partly why LISP was termed for a long time "the A.I. language". A.I. is not tied to a single language since long ago.

agents [RN95], possibly with some degree of autonomy or *intelligence*, which co-operate (or compete) in solving a task. Communication among distributed agents poses a practical problem. Although neither the concepts or technology needed to start a distributed computation[2] or to perform the subsequent communication, code and task migration, etc. are unknown or radically new, an acknowledged way of doing that easily has still not been reached. RMI and RPC give only a very basic mechanism, and a homogeneous, widely accepted proposal for transparent access to distributed data (which has a host of associated problems) is yet to come.[3] We want to point out, however, that access to data and procedure call has a pleasant symmetry in logic languages, thus reducing the number of cases to be treated.

Following on with the last bullet, there is one point worth paying attention to. If single-agent systems can admittedly take advantage from higher level languages, should it not be also the case with "multiagent" and "distributed" systems? Many widely used architectures nowadays are based on a *multilingual* concept: applications are built using several languages, using each of them for the task it is more adequate for. One possible reason is avoiding using languages for tasks they are not good at, i.e., trying to escape from language weaknesses rather than embracing language strengths. It is not unusual to use a language/tool for a task not because that language/tool is the best at it, but because others tried before performed badly. This follows the path of adopting the technology which more or less readily provides what is sought for.

As just another example of breaking Occam's razor in a more practical setting, current computer systems tend to use a high number of configuration files, each with its own syntax and capabilities. Yet all of them can be homogeneously expressed using a first order logic syntax: they are either mere facts, or, at most, rules which express dependencies among capabilities.

Without trying to dispute the practicality of the "off-the-shelf" approach, using it moves the implementation effort to the burden of making interfaces between different languages, which finally results in the definition of common interfaces (e.g., CORBA [Cro96] and COM [Rog97]), which are expected to be universal, versatile, efficient, and useful for the next future. Cannot it be the case that we are entering, at a global level, a situation similar to that of the U.S. Department of Defense which lead to

[2] Assuming, of course, that we know how to distribute this computation effectively.

[3] We even dare to say more: many widely used and appraised languages and proposals are still in their childhood, and their growing current popularity still has not been supported by the time or the design meticulousness [Han99, ABV00] needed. Absent capabilities are now being patched up by adding characteristics not present in a first design [BG00].

the definition of Ada [daC84]? While interfacing different languages is a necessity which existed from very soon in the history of Computer Science, the current trend seems to have made a virtue from this necessity, and have transformed the problem into a solution [Adl95, KA98, Mau00]. Only time will reveal if this path is the right one or not: there are confronted opinions.

1.2 Powering High-Level Programming

Restricting ourselves to the class of applications in which having a high capacity of representation is essential, and apart from the obvious possibility of multilingualism, it seems *a priori* easier to give high-level languages capabilities that are found in lower-level languages, and which are lacking from the former, than the other way around. This thesis focuses on techniques and tools to put a remedy to these lacks, and try to obtain the best from several worlds in terms of, e.g., speed, expressiveness, concurrency capabilities, distribution, and ancillary tools. A target we want to have always in sight is staying at the same descriptive level and with the same interface as the rest of the language (i.e., new capabilities readily available in other, lower-level languages should not necessarily lower the level of our language): if we do not accomplish that, we would be exchanging a (conceptual) bottleneck (using an interface to an external language) by another, different one (making our own language difficult to use, due, perhaps, to non-orthogonal, non-homogeneous constructions).

Since we have already pointed out that many of the needs and technology in what concerns (task) distribution, knowledge representation, etc. have been studied and are known, at least from a theoretical point of view, since long time ago, our contribution will be mainly practical: search for solutions to existing problems in a family of languages (logic languages, exemplified by Prolog, in our case). This search will sometimes incorporate well known technologies, or variants thereof, and sometimes we will develop new means to overcome language pitfalls or gaps. We want to point out that practically all the techniques herein reported have been put to work and proved doable and advantageous in real systems. They are not, therefore, mere ideas, but they have been tried out and evaluated in, at least, a controlled environment.

1.3 Attacking the Problem

Very possibly, an eager design of a language with all desirable capabilities would lead (assuming a successful implementation) to an efficiency so low that it would turn out to be almost useless. In fact, this has been the *casus beli* against many higher-level languages, whose practical use was avoided by extolling their merits until they are placed in the *heavenly* category of specification languages.[4] This low efficiency was fought with several weapons, among which we can highlight, in the realm of logic programming, the use of parallelism, the advances in compilation, and the removal of some language capabilities.[5]

The last one is, clearly, the simpler and handier. The second point has been, up to the moment, one of the most fruitful: there are compilers and analyzers which perform outstandingly at their task, although they progress quite slowly towards a wider spreading. The first option, using parallelism, has suffered from multiple proposals of parallel architectures and, mainly, from the intrinsic difficulty of parallelism itself; it has given very interesting results, exemplified by several implementations, execution models, and parallelizing compilers. An interesting property of parallel execution models is that they are, in principle, independent from other optimizations: a more *intelligent* compilation can be done without having to give in to abandoning using parallelism.

From another perspective, some tools to help evaluating software under development are needed. As an example, the interaction of cooperating tasks in a parallel environment, whether using shared of distributed memory, have patterns which are not only potentially different in each execution, but which are also difficult to perceive (and, therefore, to understand and improve) even in a completely controlled environment. Besides, in the quest for higher performance, it is important to find out the maximum performance throughput ever reachable by a parallel program: it may be the case that

[4]Fortunately this vision seems to be changing: as an example, let us cite the specification language **B** [Abr96], which was used to write, in about 100000 LOC, the programs which control line 14 of the Paris underground [BBM99]. This specification was proved against the properties to be met. After refining the specification using formal proofs, where many errors were discovered, 87000 lines of Ada code were generated. A conventional testing process was deployed and not a single error was found. The system is actually in production, and saved the expenses of constructing an additional line. We have here, therefore, the case of a specification language whose code produced executable programs without intermediate human intervention. Is that a mere specification language?

[5]Although written in a more concrete scenario, the following excerpt may serve as a good example of what we mean: Evan Tick writes in [Tic95]:

> *What a long, strange trip it has been! Atomic tell unification (and read-only variable synchronization) in CP was weakened into eventual tell unification and input matching in synchronization in Parlog. Deep guards in Parlog were weakened into flat guards in FGHC. Body unification in FGHC was weakened into assignment in Strand. Multiply shared variables in Strand were weakened into single-producer single-consumer (single occurence) variables in Janus, and declarative/mutable variables in PCN.*

we do not need to use more processors, or start more agents, but to redesign the whole program. From another point of view, the time savings that higher-level languages bring to the programmer are earned at the cost of a considerable run-time effort which is of difficult understanding without tools designed *ad hoc* as an X-ray apparatus which give the user a clearer image of what really happens inside an execution.

Independently from the reasons in the previous paragraphs, having a direct expression of the interaction and coordination needs in multiagent systems requires some linguistic mechanisms in order to reflect concurrency, distribution, and communication. This is but the extension of the single-process need of language with which to specify as cleanly as possible knowledge and behavior to a multiagent, distributed setting. It is also a goal, of course, not to miss a clear syntax and semantics when going from the non-distributed to the distributed case.

In view of all the above, we deem necessary having higher-level languages in which task interaction can be expressed, and to have access to ancillary tools which help in an intuitive understanding of the execution. Declarative languages are clear candidates for, at least, specifying and prototyping systems which perform complex tasks with a high degree of symbolic computation. The advantages of these languages stem, on one hand, from a simple, non ambiguous semantics, which eases the automatic analysis, and, on the other hand, from the facilities they give the programmer when allowing him/her to not to pay attention to, e.g., which kind of machine is the program to be run on, memory management details, high level representations of the data, etc.

It is certainly possible that a more widespread knowledge and use of declarative languages was limited by issues concerning their efficiency. But we think also that there are other important reasons (which we will not deal with). As an example, an incomplete teaching, which gives the impression of being inferior, less complete, or not as amenable to many tasks as other, more traditional or industrially more supported languages.

In this thesis we will study a series of techniques to cure, or at least to alleviate, the aforementioned difficulties. The points to be touched can be grouped in three categories:

Visualization and Simulation: program execution visualization has been used in Computer Science with educational aims, to debug programs, and to represent data. It becomes very important when it comes to the realm of parallel, constraint, and logic programming, due to the distance between the programmer and the execution. This distance relieves the programmer from paying attention to nitty-gritty implementation details, but, on the other hand, it makes it more difficult to understand how an execution in particular proceeded. As a drawback, optimizing a program may require some knowledge of the internal execution of the language.

Without going down to that level of depth, adequate representations of an execution can help to understand intuitively its weak points and, therefore, it gives a first hint towards a solution.

Optimizations for Parallelism in Logic Languages: parallelism has been looked upon since long time ago as a clear possibility to improve the efficiency of logic programs by exploiting the potential of multiprocessor machines. Parallel execution paradigms which do not break with the semantics of sequential execution, and which retain the view of resolution as computation, were identified. In practice, multiple parallel systems have been developed, with different capabilities and approaches to the exploitation of parallelism (essentially, or- and and-parallelism, unification parallelism, and *stream* parallelism). There are however two problems which appear recurrently, related to the complexity of parallel programming. On one hand, the memory usage, comparatively higher than in sequential execution, due not only to the fact that there are several processes running at the same time, but also related to the cost of the additional data structures needed to synchronize the different tasks. On the other hand, the additional time needed to prepare these tasks for parallel execution and to start them up. In common cases, as in recursions implementing simple/similar operations on series of elements, these delays add up. We will study both problems and we will design and test techniques aimed at ameliorating them.

Concurrency: Concurrency is, in many cases, the most natural and elegant way to express some algorithms, apart from being the best mechanism to make a better use of resources in the presence of blocking or slow calls. The interpretation of logic programming as a collection of processes (one per predicate) is well known, and proposals based on it use implicit concurrency, which quite often makes it difficult to understand the program operationally and causes a high resource consumption. Also, due to efficiency concerns, some capabilities of logic programming are removed (notably, implicit search). We will propose a new technique for communication and synchronization which uses a different concurrent semantics for accessing the database. Higher level constructions for concurrency and communication can be implemented on top of this scheme.

During the development of this thesis, visualization and simulation were used as tools to study the performance of parallel and constraint programs, and they were developed simultaneously. For the sake of clarity, we will not follow the time order, but each block will have its own separate chapter. We will now comment on each of them.

1.4 Parallelism, Concurrency, and Logic Programming: a (Very Short) Introduction [Chapter 2]

Several families of parallel logic languages have been proposed, among which we can cite those based on shared memory which address and-parallelism [Her86a, She96, TH87, CA86, Lin97, Bar94], either dependent of independent, or-parallelism [Car90, Kar92, Lus90], combinations thereof [War88, SCWY91a, GHPSC94], unification parallelism [Bar90], and systems which, belonging to any of those classes, are built assuming a distributed memory [AR97, Kac90, Kac84]. The implementations of the latter are in general quite different from the shared memory machine ones, although some proposals show shared characteristics [GP99].

The implementation of parallel logic programming systems is notoriously complex; we will review in this chapter some well-known techniques for shared memory machines and and-parallelism which try to preserve sequential semantics, and we will point out some practical problems of many parallel systems. We include in this chapter also a short introduction to proposals of concurrency in logic programming [Sha89], which quite commonly use alternative semantics. We will not deal, however, with details of their implementation.

1.5 Visualization of Sequential, Constraint and Parallel Logical Programs [Chapter 3]

Visualization has been used with different aims, among which we may highlight education, debugging (both for efficiency and for correctness) and scientific data representation. Our focus is the second of these issues, although some of the visualizations can be used with educational purposes.

As software becomes more complex, the tools needed for its development are more elaborated, and the way these tools work becomes more difficult to understand intuitively. The interaction of the different parts of an application is less straightforward, and a correct depiction can help to understand intuitively problems which could not be perceived clearly otherwise. Data and/or algorithm visualization is usually intimately linked to the data format and the control constructions of the language. Therefore, when applying visualization techniques to new programming paradigms, new graphical representation concepts are needed. In particular, the internal behavior of logic programming systems featuring constraints and parallelism cannot usually be understood immediately.

1.5.1 Visualization for Sequential Execution

Sequential program visualization has traditionally been related to the use of And-Or trees in order to reflect the traversal of branches of a tree in an execution. The usual implementation, using backtracking, uses a depth-first search in the tree.

We will not detach radically from this visualization, which is enough for our initial purposes, but we will use a variant of And-Or trees, named AORTA trees [EB88], which offer a somewhat more compact representation. Besides, we will add the possibility of knowing explicitly the origin of the instantiation of each of the variables in the execution (so that capturing programming errors which can give rise to unexpected execution results can be made more easily), and we will see how these capabilities are implemented in a tool, *APT* [CH00b, LC97]. We will also see how *abstractions* of the execution can be represented, so that efficiency problems can be studied in a better setting. In particular we will see how a parallel execution can be represented as a kind of abstraction of a sequential execution tree, in which some nodes are annotated for parallel execution.

1.5.2 Visualization for Parallel Execution

The behavior of parallel systems and algorithms is very often not trivial. Efficiency or correctness problems, both in user programs as in the system which executes them, can be located only by means of a global understanding of the whole process. Visualization tools, with their ability to offer "the whole picture", are a convenient help [Kar92, GH95] in research related to parallelism and concurrency. Parts of this thesis describes the design, implementation, and evaluation of a tool aimed at depicting the parallel execution of logic programs [CGH93, CGH92a].

This tool offers visualizations for restricted independent And-parallelism (RAP), with the possibility of task suspension, Or-parallelism, and determinate And+Or-parallelism. Such a tool has been extensively used in the study of correction of the implementation of parallel logic programming systems [HG91, Kar92] and in the subsequent improvements to the compilation and scheduling algorithms (see [HC96] and Chapter 4). The tool has been instrumental for the work related with parallelism of logic programs in this thesis.

1.5.3 Visualization for Constraint Logic Languages

The execution and machinery of Constraint Logic Languages is inherently complex, derived from the necessity of treating constraints and equations as first order elements in the language. This makes understanding the actual process executed from the beginning of the computation until a solution is reached not straightforward: the combination between the control of the program and the internal constraint solver must be taken into account. The solver determines, depending on the state of the data in each moment, which execution branches can be taken in every moment, and when backtracking (if needed) will be performed.

Control visualization can be similar to that of logic programs (which, on the other hand, should be considered as constraint programs over the domain $\mathcal{H}$ of Herbrand terms). However, visualization of data (and their relationships) in constraint programs is radically different and possibly very complex. Offering a representation thereof in an intuitive format will help to understand globally the domain narrowing as performed by the solver. That knowledge would lead to a faster program development, and would also allow an easier problem detection.

As part of this thesis we have designed and implemented a visualization paradigm for constraint logic programs which depicts data and constraints (using mainly finite domains, $\mathcal{FD}$) [CH00a]. Special attention has been paid to the use of abstractions as a means to show the evolution of the execution in a more intuitive and *compressed* fashion [SCH99].

1.6 Optimizing Executions with Data Parallelism [Chapter 4]

A sizeable part of the execution time of many programs is used in simple constructions which process data structures elementwise, aggregating its contents into a single piece of information by some sort of reduction, or returning a data structure similar to the initial one, implementing a kind of mapping. These operations and their variants are one of the main targets of the so-called data-parallelism, which tries to assign processors to every element to be treated in order to perform complex operations in constant time (in the best case) [MR90, Thi90, HQ91, Thi86, BLM93b]. Assuming independence (in the sense of Section 2.4 and [HR95]) in the computation of the different iterations, a direct parallelization boils down to generating a task for each piece of data. Data-parallelism can therefore be seen as a particular form of independent And-parallelism. This approach has the associated problem of the accumulated time in the creation of the parallel calls.

We study several ways to avoid accumulating these delays: compiling the original code so that several tasks (ideally, all of them) are started (almost) at once, unfolding partially the loops, and making a binary split of the computation on the data structure, which in turn splits and breaks up the work of starting the different tasks. Implicitly, this subdivision stems from considering the generation of the structures for the task startup as part of the work to be performed.

Not all the loops implementing an aggregation are reducible to the last format (binary execution): strictly speaking, the operations performed are not the same as in the original, sequential loop. Right and left associativity are needed in order to guarantee the same result. We will formalize the unfolding schemata previously evaluated, assuming the independence conditions are met.

It is a common case (but not strictly necessary) that the operations on each of the elements of the data structure are deterministic. Since the proposed transformation does not introduce nondeterminism, the computation as a whole would remain deterministic. As we will see in Chapter 5, this property and the particular shape of the code generated by the transformation can be taken advantage of by low-level modifications of an And-parallel WAM implementation.

1.7 Some Optimizations for Independent And-parallel Systems [Chapter 5]

Although one of the main attractions of logic programming is its implicit nondeterminism (usually implemented as a search), it is true that a great deal of applications have sizeable parts which are deterministic. Identifying these parts is interesting in that the implementation of non-determinism needs additional memory and execution time, more acute in the case of parallel processing: *backtracking* needs synchronization and workload distribution among several agents, which forces generating internal, complex data structures in order to *foresee* future backtracking scenarios.

Determinism information (either given by the programmer, or synthesized by an analyzer) can be used to produce more efficient executions by not generating some data structures. In particular, deterministic programs which are obtained from applying the transformations proposed in Chapter 4 can take advantage of the low-level optimizations. The key idea is the existence of deterministic computation subtrees. They can be executed without keeping the information pertaining to backtracking inside them (and, in fact, without performing backtracking at all), and using an adaptation of the tail recursion technique, with the associated gain in time and memory.

We have designed and implemented mechanisms which allow a more efficient parallel execution, once the deterministic calls have been identified. The experiments show a dramatic reduction in memory consumption and increased speed [PGT+96, TPGC94, Car93].

1.8 Realistic Simulation of Parallelism in Logic Programs [Chapter 6]

Predicting the behavior and efficiency of a parallel execution in several machine configurations (outstandingly, with different number of processors) is not easy in non-trivial algorithms. Even in programs with a simple, regular structure, there are oscillation phenomena when the number of processors increase. In other programs with irregular computations [Her00], the possibilities are even more involved.

Testing programs in an actual machine with the desired resources can be even impossible, because of the difficulty of accessing such machine, or because of its non existence. Simulating a parallel system is of great help in these cases, since speedups for a parallel program in a given configuration can be predicted with a reasonable accuracy. Different parallelizations of the same program can also be compared in scenarios which would be actually infeasible (e.g. with an unbound number of processors) in order to estimate how good is the parallelization. The efficiency of an implementation can also be compared with the ideal one in different scenarios, by introducing (or removing) delays in selected parts of the execution, using more/less efficient scheduling algorithms (using, e.g., an *a posteriori* global scheduling), etc.

Similar experiments, based on a program metainterpreter [HC91], had already been carried out. We will, however, use program traces initially designed for program visualization and reinterpreted as descriptions of an execution graph which can be rescheduled. The results obtained with a tool which simulates parallel execution using those traces are highly concordant with those obtained from real executions (up to the point a real execution can be performed: physical limitations in the validation show up in this case), which indicates the usefulness of the tool in order to predict its behavior in executions impossible to realize in practice [FCH96].

1.9 Concurrency in Prolog Using Threads and a Shared Database [Chapter 7]

Concurrent algorithms do not necessarily result in programs with a good parallel behavior. Concurrency is, however, a natural way to write certain algorithms and to design software architectures with some degree of complexity—e.g., agent-based systems, where planning and knowledge revision are induced by external changes (messages from other agents, sensor stimuli...) and the moment in which they happen is not explicit in an algorithm: only the causal relationship between processes is specified. We can say that parallelism is a physical phenomena, while concurrency is a programming abstraction [Smo95].

We can roughly distinguish among explicit concurrency (when specifying which parts of the program are to be executed concurrently is needed) and implicit concurrency (when all the calls are concurrent by default). A great deal of proposals of concurrency in logic languages fall into the latter type [Sha89], synchronization being left to the instantiation of variables, although later approaches [JH91] added constructions aimed at encapsulating sequential computations inside a general concurrency setting. Concurrent logic languages have traditionally trimmed different capacities [Tic95] (as, for example, search in the presence of concurrency), arriving at the concept of *committed-choice languages*.

A practical problem found in concurrent logic languages is the high number of tasks, quite often unnecessary, and the difficulty of having an explicit control on the execution flow, which is sometimes necessary. Due to that, some modern languages which come from a tradition of logic and constraint programming (although maybe not completely representatives of it) and which had initially explicit concurrency [Smo94, Smo95] have switched to a sequential-by-default execution model [HF00].

We propose a set of basic thread creation primitives which assign an execution thread and a memory space to each concurrent goal, and do not affect directly the variables of other goals. The memory space is separate in that variable unifications are local to the scope of the concurrent goal. From a practical point of view, this facilitates encapsulated backtracking (i.e., backtracking restricted to the concurrent goal) and independent garbage collection in each thread. Space independence is currently achieved by making a fresh copy of the goal to execute to a new WAM. Backtracking is taken care of by providing builtins which cause a failure (and subsequent backtracking) in an already existing concurrent goal.

Communication among concurrent goals is based on using the (shared) dynamic database, visible by all the threads. Accesses to the database in a Prolog system usually fail if the fact does not appear in the database, and updates usually follow the "logical vision" [LO87]. This behavior is not appropriate for our aims (we want database updates to be immediately visible to other threads), and we have based our proposal in the similarity between the access to the database and the access to a tuple space, such as that used by Linda [CG89b, CG89a]. This similarity suggests augmenting the database access with an operational semantics for concurrency: the access to the database is atomic, and calls without a matching fact suspend instead of failing. Suspensions and resumptions allow implementing synchronization among tasks; the data added to the concurrent facts can be used to perform data communication. This communication means has a higher cost than a regular call, and therefore should be reserved to tasks with high granularity if speedup is sought for.

We include examples of the use of the shared database, along with experimental results aimed at evaluating the efficiency of task creation and data communication. They support the flexibility and expressiveness of our communication scheme, and confirms the possibility of achieving speedups with a right selection of the granularity level. We show also how higher-level concurrency and parallelism schemes can be implemented using the shared database and explicit thread creation.

1.10 Programming Systems Used in the Thesis

The work reported in this thesis spans several years, and the platforms used in it changed throughout this time. The initial implementations, focused on parallelism, were made in the *&–Prolog* system [HG91, Her86a], and some of the results have since made their way through to other academic and commercial platforms, such as ACE [PGH95, PG95a] and SICStus Prolog [Swe99]. The first visualization systems for parallel execution were implemented using directly X Window and Athena Widgets. Those for visualization of sequential execution and constraint logic programming used Tcl/Tk and VRML.

More recent work, related to concurrency, has been carried out in the Ciao Prolog system [HBC+00, HBC+99, BCC+97]. Experience gained in the *&–Prolog* platform is nowadays being transferred to Ciao Prolog.

1.11 Thesis Development

As the work in this thesis was being developed, new necessities were identified and solutions for them sought. The quick development of Computer Science and its increasing presence (both regarding the available technology and what society demands from it) has made some points more interesting than others. Techniques initially thought to be difficult or impractical to work through have been found not only practical, but even advantageous and necessary. This is due to, e.g., the improvement of hardware and communications, and ideas elaborated at a time when these developments could not have been foreseen, were finally phased out.

The introduction of new technology, and the shift in what computers are demanded for, is intimately related with the changes in the way programming is performed. The present research work has followed these changes, and its focus has moved in order to adapt to new problems as they were recognized as important. It should, however, be pointed out that the problems initially tackled have neither been completely solved nor invalidated by the time: many of them have just been procrastinated, and the ideas shown here remain valid.

1.12 Contributions and Thesis Structure

The main contributions of the thesis in the three fields we will tackle can be summarized as follows:

- Contributions related to the visualization of logic programs:
  - Visualization as a graphical representation of a series of observables which describe an execution at a given abstraction level.
  - Definition and study of a paradigm of visualization for sequential programs, based on [EB88], and design of an extensible tool which implements such paradigm.
  - Proposal of several visualization abstractions for the sequential case aimed at representing computations of average and big sizes.
  - Study of a visualization paradigm for parallel execution and assessment of its usefulness in real cases.
  - Design of methods to represent the evolution of constrained variables, and implementation of a tool which allows depicting them.
  - Definition of a generic method to visualize constraints among variables using a representation of their values.
  - Proposal of an abstraction to visualize constraints based on the constraint net, and proposal of a 3-D abstraction to visualize constrained variables based on depicting the size of their range.
- Contributions related to the parallel execution of logic programs:
  - Study and evaluation of transformations for parallel programs aimed at alleviating the delay due to the preparation and launching of parallel tasks.
  - Application and evaluation of these techniques to the case of programs using lists and vectors.
  - Formalization of the transformations and identification of sufficient conditions for their applicability.

- Study of the effectiveness of automatic specialization in the programs which result from the previous transformations.
- Proposal and experimental assessment of modifications in an abstract machine for parallel execution aimed at increasing speed and decreasing memory usage in a wide class of programs (including many which result from the previous transformations).
- Extensive study of the adequacy of a simulation method for the prediction of execution speed in a parallel program.

• Contributions related to logic programs and concurrency:

- Design and implementation of a set of builtins which allow viewing a goal as a first-order object of the language, making it possible to start its execution in a separate environment, wait for its termination, ask for more solutions, etc.
- Application of the aforementioned builtins to the case of concurrent goals, allowing the execution of goals in separate threads.
- Identification of the similarity between the access to the local shared database and to a Linda blackboard, and proposal of a semantics to access the database where suspension replaces failure.
- Implementation of a method to access and update the database atomically, giving a means to synchronize and communicate data among different threads.
- Evaluation of time and space efficiency of the aforementioned implementations.
- Implementation and evaluation of different proposals of concurrency and parallelism based on the previous ideas.

The body of this thesis is composed of several selected papers, written in collaboration with other coauthors. All of them have been published in national an international journals and conferences with peer-to-peer review. We want to clarify the sources where the chapters came, and whose their authorship is:

• Chapter 3 is based on the following papers:

- [CH00b, CH00a], written together with M. Hermenegildo and included as chapters in the book resulting from the *DiSCiPl* ESPRIT project [DHM00],
- [SCH99], written with Göran Smedbäk and M. Hermenegildo, and presented at the *Practical Applications of Constraint Logic Programming* conference,

  - [CH98], written with M. Hermenegildo and presented at the *Appia-Gulp-Prode* conference,
  - [CGH93], written with L. Gómez and M. Hermenegildo and presented at the *International Conference on Logic Programming*, and [CGH92b], presented at a workshop of the same conference.

  The implementation of the visualization for sequential execution and for constraint programs were made as Master Thesis (under the tutoring of the PhD candidate) by Ángel López Luengo (*APT* [Lue97]), J. M. Ramos (*VIFID* [Ram98b, RC98, Ram98a]), and Göran Smedbäk (ProVRML and *TRIFID* [SCH99]). The implementation of *VisAndOr* [CGH93] is based on a previous one (*VisiPal*) by M. Hermenegildo and R. Nasr [HN90], and was made in its greatest part by L. Gómez, M. Hermenegildo, and the author of the thesis.

- Chapter 4 is based on [HC96], written with M. Hermenegildo and published in the *Computer Languages Journal*, and on [HC95], presented at *EuroPar*. This paper has been improved with a formalization section not present in the original work.

- Chapter 5 is a selection of [PGT+96], written jointly with E. Pontelli, G. Gupta, D. Tang, and M. Hermenegildo, and published in the *Computer Languages Journal*. The sections which I consider to be in its majority a work of the rest of the authors have been removed for the present thesis.

- Chapter 6 is based on [FCH96], written jointly with M. J. Fernández and M. Hermenegildo, and presented at *EuroPar*. The implementation of the simulation program was mainly done by M. J. Fernández as Master Thesis.

- Chapter 7 is based on [CH99], written with M. Hermenegildo and published at the *International Conference on Logic Programming*. Part of the ideas there appeared also in [CH96].

Chapter 2

Parallelism and Concurrency in Logic Programming: a (Very Short) Survey

Chapter Summary

This chapter gives a brief introduction to parallelism and concurrency in Logic Programming, focusing mainly in and-parallelism. The concept of goal dependency is informally reviewed, and we introduce one of the techniques more widely used to implement restricted and-parallelism. We then go over some efficiency problems which appear in parallel executions and which are specially relevant in some types of programs. We finally point out how we approach ameliorating these problems.

We then make a similar review with concurrency. Although in some ways related to parallelism, concurrency can be explained and taken advantage of without parallelism, and, in fact, it has a wider range of practical applications. We will see some problems associated with concurrency in logic languages, and how different proposals have tried to overcome these drawbacks. We finally give a short account of our proposal for communication and concurrency.

2.1 Parallelism in Logic Programming

Much work has been done up to date in the study of parallel logic program execution. There is ample literature on the matter, and a sizable number of implementations, both for shared and distributed memory machines, have been realized [GC96]. Shared memory systems have been more successful so far, due to the easier programming interface offered by this model[1] and to the better speed-ups which can usually be obtained thanks to the smaller communication cost. Some implementation models are based on adapting

[1]This decision is nowadays supported by the increasing availability of high-end shared memory multiprocessors at a relatively low price. Modern stock distributed memory hardware, however, can change the scenario.

sequential abstract machines for the execution of Prolog in order to separate clearly the execution of the different parallel goals. In this case, several sequential executions proceed at a time on different memory zones, trying to minimize interaction and, therefore, the need of communication. This proposal, which quite often uses the *marker model*, has produced the more efficient implementations, since most of the optimizations already present in sequential machines are reused and put to work.

In particular, many and-parallel models are derived from the RAP/WAM marker model, introduced in [Her86a, Her87, Her86b], and which appears later, with refinements, in [HG90, She92b, SH94, GHPSC94, PG97]. This model has shown to be practical and capable of giving good speed-ups in parallel execution, and it has influenced the implementation of other systems featuring and-parallelism. This initial model can however be optimized in order to reduce the costs associated to parallel execution in common cases and to give a better overall efficiency. These optimizations are also general enough as to be used in other implementations of and-parallelism using the marker model.

In the next section we will give a quick account of different schemes for parallel execution in logic programs, restricting ourselves to the models which we will be dealing with in the thesis. A more thorough survey of the topic can be found in, e.g., [GC96].

2.2 Types of Parallelism in Logic Programs

Parallelism in a logic program can be, basically, of two different kinds: **and-parallelism** and **or-parallelism**.[2] Let us consider the following example [Her86a] which determines the crew needed for a flight:

```
crew(t(X, Y)):-                   navigator(peter).
    navigator(X),
    pilot(Y).                     mechanic(peter).
crew(t(X, Y)):-
    mechanic(X),                  ...
    pilot(Y).
```

Figure 2.1 shows a search tree for the query

```
?- crew(t(peter, Y)).
```

[2]There is also the possibility of performing parallelism in during each head unification [Bar90], which we will not deal with.

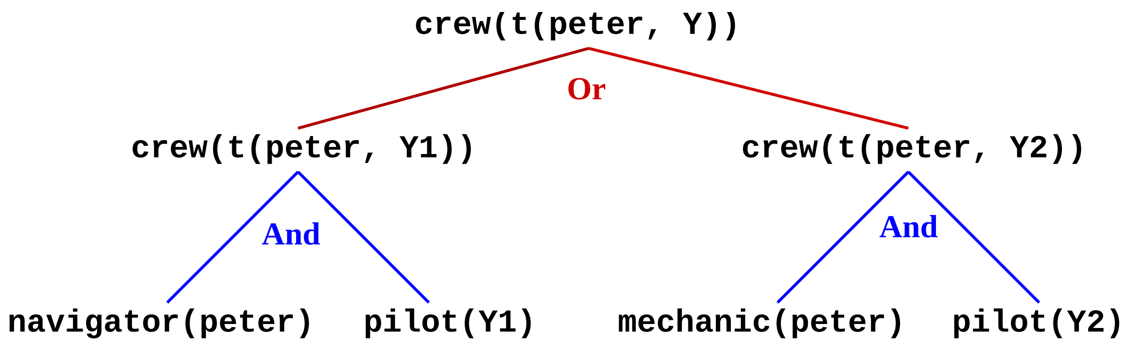


Figure 2.1: Or- and and-parallelism in logic programs

where we can see that there are two alternative search paths which satisfy it. The existence of two clauses for the predicate `crew/1` makes it possible to explore simultaneously the two subtrees which root at the predicate call. This gives rise to the so-called or-parallelism, in which there are no data dependencies (at least in principle), since alternatives can be thought as being conceptually independent from each other. In practice, dependencies can appear due to side effects (e.g., assertions in the data base, file operations, ...) in different branches of the search tree, and also at an implementation level, due to variables appearing in several branches which could be instantiated at a time by several processors.[3]

Or-parallelism can be primarily exploited in code which has non-deterministic search. Techniques to implement or-parallelism while respecting completely the standard Prolog operational semantics for the sequential execution are well known [GSC92, War87b, Lus88, Kal87a, Hau90, Sze89, War87a, CH83, Kar92, Car90]. Some commercial systems [Swe99, ECR93] include or-parallelism.

Let us consider the execution of one the two alternative clauses, e.g., the first one. In order to satisfy `crew/1` we need to satisfy both `navigator(peter)` and `pilot(P)`. Both goals can be executed simultaneously without the need of communication, since no one of them needs information from the other one. The parallelism which results from the simultaneous execution of both goals is termed and-parallelism. And-parallelism can

[3]At the risk of overloading the terminology, we will use the term *processor* to refer to any entity able to execute code, which could more appropriately be termed *agent*, as is done in [Her86a, HG91]. We however run the risk of being misinterpreted as referring to intelligent agents [RN95], when out aim is rather denoting processing units. The processors we refer to do not need to be actual processors, in the physical sense, but they can as well be processes in multiprocessing operating system. It is clearly necessary to have different physical processors if speedup is sought for; this should be implicitly assumed when we study efficiency and speed matters.

be used both in deterministic and in non-deterministic problems, and hence it has, in principle, a wider range of applications than or-parallelism—which does not imply that better speedups are to be obtained. However, the implementation of and-parallelism is a delicate task due to the interactions of the goals which are to be executed in parallel.

In general, while sequential execution keeps *active* only one branch of the search tree (from the root to the node being worked on), parallel execution proceeds at once in several nodes of the tree, which forces maintaining several parts of it active at the same time.

2.3 And-Parallelism and Data Dependencies

A general principle when writing parallel programs is to try to minimize synchronization and communication among processors. This is of course applicable to Logic Programming, and it should be taken into account in order to have a good parallel execution, which can take longer than the sequential execution of the same program otherwise [HR95] (even assuming that the time used in the synchronization among processors is zero).

Let us allow all goals in a clause body to execute in parallel without any restriction other than the need of correction (i.e., equality of observable results: same solutions and in the same order) with respect to the sequential execution. Let us consider also the following pure implementation (no side effects, no metalogical predicates) of the `pilot/1` predicate

```
pilot(P):-
    license(P),
    medical_permit(P).
```

Assuming that the initial query is `?- pilot(peter)`, a sequential execution would call `license(peter)` first and, if it succeeds, `medical_permit(peter)` would be executed next. The parallel execution would try to fulfill both goals simultaneously. If the sequential execution was successful, the parallel one will succeed also, and in a shorter amount of time, since there is no additional overhead for task communication, and the time taken by the conjunction should be at most that of the larger task, instead of the sum of both goals, as in the sequential case.

If the query to post is `?- pilot(P)`, then a sequential execution would first call `license(P)`. The variable `P` could be unified with the constant `peter`, and, after that, `medical_license(peter)` would succeed. However, if both goals are executed at a time, their execution can yield inconsistent results for the variable `P`: the order of solutions of `medical_permit(P)` might not give `P = peter` as first solution. One of the goals should be then re-executed in order to generate new solutions and reach to an consistent agreement. This re-execution was not necessary in the sequential program, and can, *a priori*, cause more work in the parallel case. Moreover, re-execution could change the order of solutions with respect to the sequential execution, which in many cases is not desirable, as it violates the "observable semantics".

The source of this difference is the restriction the first goal imposes over the search tree of the second one by means of the instantiation of a shared variable, which does not happen in the parallel case, where there is no search space preservation. This problem appears also even if variables do not appear textually in two goals: let us consider the second clause of the predicate `crew/1`:

```
crew(t(X, Y)):-
    navigator(X),
    pilot(Y).
```

A query such as

```
?- crew(t(Z, Z)).
```

would lead the same problem, since `navigator/1` and `pilot/1` will share variables at run time.

The search space preservation problem can be restated as a notion of independence of the computations which can in many cases (e.g., when only non-side-effects goals are being considered) be described in terms of independence (using variable aliasing / sharing) of the parallel goals [GadlBHM93]. Or-parallel execution have this independence granted since the computation in different branches is already logically independent.

2.3.1 Dealing with Dependencies

A method to overcome the dependency problem would be to generate all solutions for every goal and make a cross-product of the consistent solution, so determining the set of solutions for the clause [RK89]. The main advantage of this method is its ease of implementation, and its main disadvantage is the inefficiency,[4] both due to the amount of work performed (let us think of a case where only one solution is needed: we would generate

[4]Leading, in some cases, to non-termination!

all of the solutions), and the high cost associated to the removal of inconsistencies and to the memory needed to store intermediate results.

A more practical method consists of determining the dependencies among the goals in the clause body. These dependencies can be used to guide a method of resolution with parallelism which avoids redundant work. Once these dependencies have been determined, each model of and-parallelism differs in the effect that they have on the number of goals to be executed in parallel and on the communication among them. In some sense, the generate-and-cross-solutions strategy corresponds with a total lack of knowledge of the dependencies of parallel goals.

Several solutions to the problem of determining dependencies have been proposed [Con83, CDD85], but the method which has finally shown more successful is the one based on DeGroot's method. DeGroot's *Restricted And-Parallelism (RAP)* finds a compromise in the dependency determination of the goals in a clause, splitting the work between compile time and execution time. The proposal is based on generating several possible dependency graphs at compile time, corresponding to the different call modes. These graphs are combined in the so-called *Execution Graph Expression (EGE)*. The run time instantiation of the arguments determine which goals can be run in parallel and which ones cannot. It also cuts down the parallelism relationships among objectives to (nested) fork-joins.

This achieves, on one hand, a considerable simplification of the executing machinery, and, on the other hand, a considerable widening of the number of cases in which parallelism can be detected (dependencies between two goals of a clause did not impede other goals of the same clause being executed in parallel). This solution, however, lacked accuracy and expressiveness. It did not provide an specification of the procedure to follow when one of the parallel goals failed, and it did not give an indication of algorithms or heuristics to generate the EGE (it is quite common that there are several different EGEs for the same clause). Additionally, the complexity of the tests to perform restrained its performance.

2.4 Independent And-Parallelism

Many of the solutions to these problems were provided by Hermenegildo [Her86a, Her86b, Her87, HN86] with the development of a simpler and more powerful set of expressions of execution graphs named *Conditional Graph Expressions (CGE)*, and with a complete operational semantics for a type of parallel execution of logic programs. This version of RAP was termed *Independent And-Parallelism (IAP)*, and an implementation based on the *Warren's Abstract Machine (WAM)* [War83] was proposed. The name *independent* comes from one of its main characteristics: once the dependencies among goals are established, only those which do not share variables at run time can be executed in parallel. This ensures that the results are not inconsistent [HR89].

This approach guarantees that a parallel execution will not be slower than a sequential one, so that the complexity expected by the programmer remains unchanged, but performance is increased thanks to the larger number of tasks being executed in parallel. The absence of shared variables simplifies synchronization, which only has to deal with control issues, since there will be no *competence* for variable binding. This eases the design and implementation of the execution model, and allows having an operational semantics compatible with that of Prolog.

2.5 Dependent And-Parallelism

There is another type of and-parallelism in which parallel execution is allowed among goals which share variables. This kind of execution is called *dependent and-parallelism (DAP)*. There is an interesting taxonomy of DAP approaches in [PG97], where the maximum efficiency attainable by some primitive operations of a system is used as classification criteria. In this thesis we will restrict ourselves to higher level descriptions, guided by the observable semantics of the language.

DAP encompasses IAP, and therefore it gives rise to more parallelism opportunities, and it should in principle be able to obtain speedups in a wider range of programs. A way to perceive this type of parallelism is to simply consider that the independent operations are happening at a smaller granularity level, i.e., at the variable binding level, instead of at the resolution subtree level.

A disadvantage of DAP comes from the complexity if its implementation. On one hand, forward execution must classify goals as consumers or producers of variables (a classification which changes dynamically), apart from making sure that there are no low-level conflicts among different bindings of variables (either by locking the access to the structures being constructed, or by other, more sophisticated schemes [PG97]). On the other hand, a correct backtracking which abides by sequential Prolog semantics is not easy, since it may require stopping other agents while they are making forward execution, or, at least, coordinate several parallel backtrackings until a "safe" point is found.

As a result, many systems which adopt this type of communication among tasks eliminate backtracking altogether, and therefore also non-deterministic search of the type *don't know*, i.e., those which allows finding all the solutions to a problem in a "builtin search" fashion. *Don't care* non-determinism is used instead, which assumes any branch chosen at a time is the correct one, and execution continues with it.[5] A branch is considered selected when a sequence of designated goals, termed guard, succeed. The guard usually involves simple primitives which do not generate bindings, but which use pattern matching or simple tests on the status of the variables. Each clause has a guard, and execution suspends until there are bindings compatible with the guard(s) of some of the clauses. At that moment, one of the compatible clauses is selected and execution continues in the body of that clause, where new bindings can be generated which can wake up other suspending clauses. Committed choice disallows implicit search, and one of the most interesting characteristics of logic languages is therefore lost: search has to be explicitly programmed. Parlog [CG86], Concurrent Prolog [Sha83], and GHC [Ued86] are well known examples of committed choice languages. We will come back to this matter in Section 2.10.1.

Among the proposals of DAP which feature backtracking we can cite the scheme based on Attributed Variables [HCC95] and the DDAS scheme [She92b, She92a, She96]. The latter has Prolog semantics, which is implemented by classifying each variable either as producer or as consumer based on the their position in the search tree. Although the range of applications in which DDAS can execute in parallel is larger than in IAP, the complexity of its implementation, such as the one in DASWAM [She96] makes it relatively slow in the sequential parts. Other, more recent proposals, such as the *Filtering Binding Scheme* [PG97], which can be seen as an instantiation of the DDAS execution model, are expected to outperform DASWAM.

There are other alternatives which, although we will not deal with in this thesis, deserve being mentioned. One of them follows the Andorra Principle proposal: give preference to deterministic calls, and suspend the non-deterministic ones until they become deterministic, or until they have to be evaluated because there are no deterministic goals available. This principle can be used to guide parallel execution while allowing a *don't know* semantics: DAP is allowed only among deterministic goals, and the non-deterministic ones are reduced by a single processor. This is the approach followed by Andorra-I [SCWY91c, SCWY91b]. The operational semantics does not follow Prolog's, which makes Prolog programs to have to pass through a quite complex preprocessing before they can be executed in Andorra-I with the same results.

[5]There is a more profound reason for that distinction: concurrent systems are usually reactive (data computed by ongoing computations affect the rest of the universe), and so a failing path, which in a Prolog-like language does not give an observable outcome (modulo side-effects), would change the state of the overall computation.

The Extended Andorra Model follows the same general idea as the Andorra Principle, but additionally non-deterministic computations can be encapsulated in *boxes*. The bindings generated by these computations do not propagate out of the box until they become deterministic (i.e., they cannot be changed by backtracking in the boxed computation), and at that point they are promoted out of the box. Perhaps the best known implementation of this model is the AKL language [JH91, Jan94], but other proposals have been made, such as Pandora [BG90] and ANDOR-II [Tak92].

Finally, let us point out that there are other proposals which combine IAP and DAP with or-parallelism [GJ89, GHPSC94, PGH95, GH92, GSCYH91]. They have added difficulties over and-parallelism.

From now on we will focus on IAP, and we will suppose that a preliminary analysis of data dependencies has been performed, so that we know which conjunctions can be executed with IAP.[6] In order to make clear the independence condition (both for the reader and for a final compiler) it is customary to use the "`&`" operator instead of the comma ",". Assuming run-time independence between `X` and `Y` we would write a previous example as:

```
crew(t(X, Y)):-
    navigator(X) & pilot(Y).
```

and execute it while keeping the same declarative semantics.

2.6 Shared Data Structures and Parallel Execution

Since different processors are simply executing parts of the same computation and cooperating in the construction of a solution to the initial query, it is necessary that each partial subtree solution is available for the rest of the processors. In a distributed memory machine this usually increases the data traffic, which affects performance negatively. In a shared memory machine this is solved just by using different memory zones for each goal, assigned to a other processor each, and reachable by all the other processors. Figure 2.2 shows an scheme of the execution of the goal `crew(T)`; the initial call and every of the subcalls are assigned to different processors (`P1`, `P2`, and `P3`), and the variables must be reachable by the different processors. In particular, `P1` must see the bindings made by `P2` and `P3` for the names of the pilot and navigator.

[6] If parallel execution is not possible in all the cases, we will assume that we know the conditions of the data which allow such an execution, and that we can annotate the program with *if-then-else* constructions in order to note them.

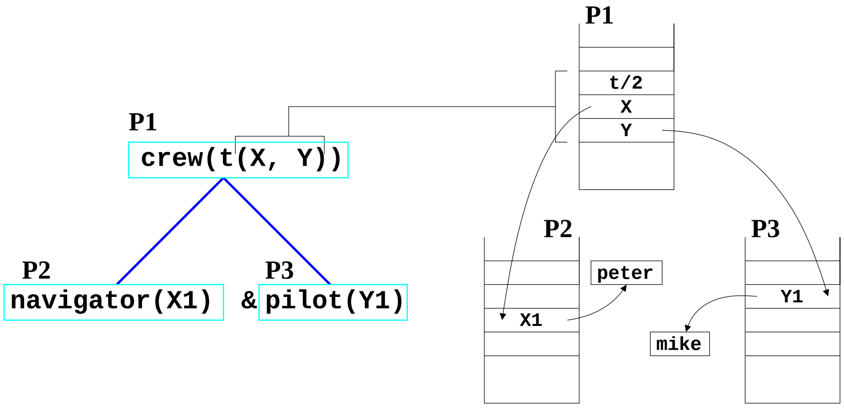


Figure 2.2: Shared data structures are needed

Not only Prolog-level data (i.e., the low-level implementation of Prolog terms) have to be shared. There are other data structures, needed for a correct synchronization of the execution which are not seen at the level of Prolog source code. As an example, in

```
crew(t(X,Y)):-
    navigator(X) & pilot(Y), ...
```

the execution needs to wait until both `navigator(X)` and `pilot(Y)` have finished in order to continue with the call to `crew(t(X, Y))`. This synchronization, both in forward and backward execution, needs additional data structures which complicate the internal machinery of the parallel system. We will describe them and their rôle in some more detail. We will assume a simple marker model for the implementation, and we will see very briefly an implementation scheme for forward execution in an IAP system. We will deliberately not describe the basic model [Her86a] and its subsequent improvements [SH96, Pon97] in depth. Scheduling issues will not be discussed, since they are not strictly necessary for this account.

2.7 Forward Execution

When the execution finds a parallel conjunction, the subgoals in it are prepared to be executed by the free processors. Execution continues after the conjunction only when all the subgoals of a conjunction have finished. The execution can then be divided in two phases: the *internal* one, when the conjunction is being executed, and the *external* one, when the conjunction has finished.

Some additional data structures are needed in order to record the status of the execution at each moment, and to mark which parts of the tree are being explored. The

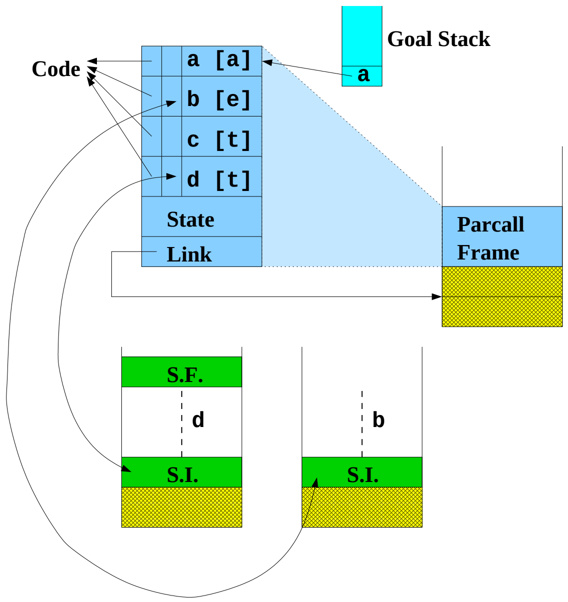


Figure 2.3: Additional data structures

main data structures [HG91] are the goal stack, the parallel call frame (*parcall frame*) and the frames which mark where every goal execution starts and ends in a stack (goal markers), represented in Figure 2.3.

When a parallel conjunction is reached a parcall frame is created, which contains the information pertaining to the goals to be simultaneously executed. In particular, that environment has:

- An entry for each goal in the conjunction, with information about that goal (e.g., where the predicate code starts) and about its state: if no processor has started executing it (`[a]`), if it is being executed (`[e]`), if it is finished (`[t]`), if it has left or not pending alternatives, etc.

- Global information about the status of the parallel call (e.g., number of goals left to finish, connections with other parallel calls, etc.).

The next step is to allow other processors to take the subgoals of the parallel call. The subgoals are left in a shared memory area which all processors can access.[7] As an example, each processor can be instrumented with a goal stack where the generated

[7] As an optimization, the processor which generates the parallel call can reserve a goal for itself, so saving passing through the general mechanism. We will see later (Chapter 4) how this simple optimization gives noticeable speedups.

goals are pushed, and from where all the processors can pick up [Her86a]. Goals in the stack have pointers to those fields in the parcall frame which have to be consulted and/or updated during the parallel goal execution.

The execution of a parallel goal taken from a goal stack starts by pushing a goal marker in the execution stack.[8] This marker is needed for several reasons, outstandingly:

- It links all the parallel goals in the correct order. In particular, it contains pointers to the corresponding parcall frame in order to access it and consult or update its fields whenever needed. Apart from that, it behaves much as a choicepoint with the peculiarity that in case of being reached on backtracking, special actions are performed in order to organize backtracking in parallel.

- It divides the stack in segments corresponding to the different goals, which helps in performing garbage collection.

Figure 2.3 shows a state in which a parallel call `a & b & c & d` is being executed. Goal `a` is not yet taken by any processor, goals `c` and `d` have finished, and goal `b` is being executed. The execution stack of `d` is shown: a *start marker* and an *end marker* signal the memory space used by this execution. `b` has not been finished, and thus it has just a *start marker*. The markers are linked in order to give an ordering to the different execution segments.

The processor which finishes the execution of a parallel goal tests (in the parcall frame) if it was the last one to finish. If so, that very same processor picks up the continuation of the goal conjunction. In any other case, a processor which has finished a goal and does not have to continue the conjunction can search for more goals to run.

2.8 Backward Execution

The term "backward execution" denotes the steps to be performed after a (logical) failure in the execution. In a system with and-parallelism, which only explores an or-branch at a time, this implies making backtracking up to the point of the logically previous choice point. No matter what the backtracking ordering is, the different calls can have been scattered over several memory zones and executed by different processors. Let

[8] Different proposals as to which stack frames and markers should be pushed onto exist. We will assume that markers go to the choicepoint stack, and frames to the environment stack (which are the same on a single-stack implementation), although this distinction is not crucial for our description. Creating *ad-hoc* stacks is also possible.

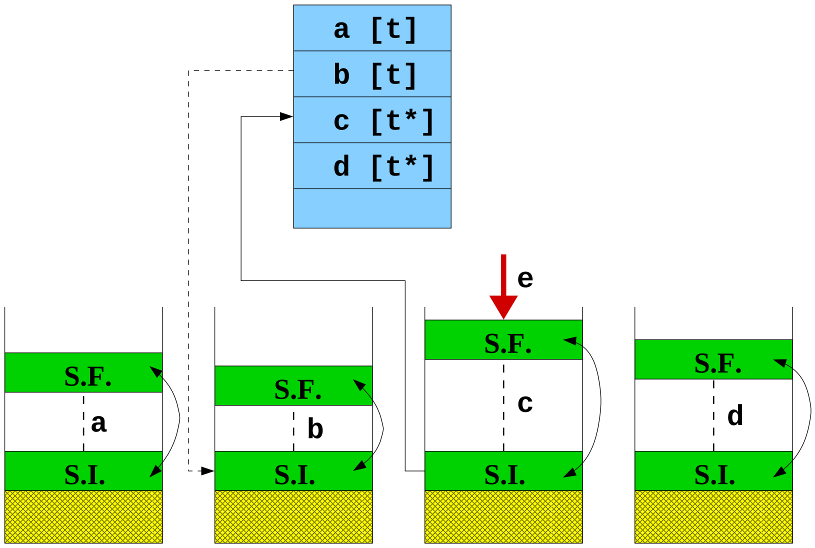


Figure 2.4: *Backtracking* on a parallel conjunction

us suppose that that we want backtracking to be performed following the sequential execution order, which preserves solution order.

There are two different backtracking cases: external backtracking, when it happens after having finished successfully a parallel conjunction, and internal backtracking, when one of the parallel calls fails before obtaining at least one solution for all the conjunction. Due to independence considerations, these two cases can be treated differently: in particular, with the search space preservation assumption, an internal failure before any solution is obtained can immediately cause failure of the full parallel conjunction [Her86a].

Let us consider the call `a & b & c & d, e`. We will assume, for efficiency reasons, that the processor which finished the last parallel goal started the execution of `e`, and in the same stack as `e`. Then, the processor making backtracking over `e` detects it is reaching a parallel conjunction because it finds an *end marker* instead of a choice point. In general, the end marker does not have to correspond to `d`. We want, however, to force backtracking over `d` first — if it has any choices left. In any case, when the parallel conjunction is reached on backtracking, one of the goals in it has to be restarted.

Let us assume that `c` was the last goal to finish (Figure 2.4). The memory segment where the rightmost goal with pending alternatives was executed (`b`) is found through the parcall frame. `c` and `d` are marked as having no alternatives, and thus a new search for `b` is started while, at the same time, `c` and `d` are restarted from scratch. A parallel execution with a smaller number of goals (`b`, `c`, and `d`) is then started.

Internal backtracking happens when a parallel goal hits a start goal marker on backtracking before any solution has been reported by the goal. If independence is maintained among parallel goals, then there is no solution for the parallel conjunction (at least with the initial instantiation). Therefore, the ongoing parallel computations can be stopped and the and-parallel goals as a whole made fail. This implements a form of intelligent backtracking.

In both cases the number of possible scenarios of (parallel) backtracking is larger and considerably more complex than in sequential execution. In general, it is necessary to have information enough as to "recover" an state which could have never happened before, and restart an execution from it. This is made more difficult by the spreading of the execution across several processors, which needs additional coordination.

2.9 Limits of Parallel Execution

There are some overheads associated with parallel execution, which can be summarized in the following points, common both to and- and or-parallelism:

- The tree structure has to be kept explicitly (at least for the *active* part of the execution tree), while in a depth-first sequential execution only a stack is needed. Creating and keeping the tree structure is an additional overhead.

- Tree traversals are needed in order to, e.g., find out which part of the tree has to be re-executed after a failure. These traversals, which are easy to do in a sequential execution thanks to the stack representation, impose some additional work to take into account.

- Synchronization among processors is needed. The data structures used for the synchronization, and the synchronization itself, can be quite complex and add an additional overhead to the system as a whole.

The sources of overhead just mentioned do not affect equally every implementation of parallel logic languages: as an example, different scheduling algorithms in or-parallel languages show a wide variation in what regards efficiency and overhead imposed to the system. This selection impacts not only the time to reach a solution (which is, *a priori*, unknown), but also the global behavior of the system: there are differences in the work which is necessary to position the agent in the node where a branch starts, the difficulty of the treatment of side effects, etc. In a system with and-parallelism all and-parallel tasks should, in principle, be executed, so the selection is less critical. Trying to take advantage of semi-intelligent backtracking in the case of and-parallel search would lead to a selection of schedulers which, unlike in the or-parallel case, prune the search tree by selecting the failure branches first. Again, in absence of more information, this selection can only be based on heuristic rules.[9] Interestingly, this has clear connections with labeling in some constraint systems.

We will see how these overheads are relatively important in some kinds of programs featuring IAP, which have code patterns which appear quite frequently, and which should therefore be studied with special attention. Although IAP limits communication to the synchronization points at the beginning and the end of parallel tasks, and scheduling overhead can be kept small, there are many possible causes of delay in every parallel execution. Solving some of them requires keeping the work local (i.e., avoiding the intervention of the scheduler) and do not perform useless work in advance (e.g., try to foresee which backtracking scenarios cannot happen, and do not make provision for them). In particular we will examine the following points:

- Preparing tasks to be executed in parallel takes some time: starting a goal in parallel is usually slower than calling it sequentially [HC96, HC95, LGHD96, DJ94]. This is because setting up a parallel task must prepare a *closure* of the task so that its execution can start in an environment different from that the goal first appeared. It is possible to identify cases where several parallel tasks can start at once, or distribute the work of preparing and starting up tasks among several processors.

- As we saw, respecting the sequential semantics has an additional cost: the possibility of backtracking among parallel calls (even with the simplification offered by their independence) adds an additional memory and time overhead because of the need to prepare the data structures needed for the backtracking [Her87, SH96, Car93]. There are, however, frequent cases where this backtracking will never be produced, and identifying them can lead to savings in the creation an maintenance of such data structures.

- In any case, for some initial data and for a given program parallelization, a fixed number of parallel tasks are generated. The fact that only some of these tasks can effectively be executed in parallel (due to task dependencies and to the program structure) limits the speedup attainable, independently from the number of physical processing units available. Understanding whether this limit has been reached can be eased with the help of simulation and visualization tools, which help to:

[9] Unless external information is available. Such information can include expected number of answers, upper or lower complexity bounds [DL93, LGHD96], ...

– Evaluate if a given program cannot accelerate more because of its internal structure: delays due to scheduling, implementation overheads, and unavailability of physical resources can be eliminated in a simulation.

– Visualize which scheduling problems have appeared in the execution. The shape of the execution frequently gives relevant clues.

– Study how alternative scheduling algorithms influence the quality of the execution.

Attacking the previous points will help to increase efficiency, both in time and memory consumption. These points are not completely independent: as we will see, the conditions to improve one of them are, quite usually, necessary to improve some of the others. Simulation and visualization tools give a more intuitive support to the numerical data and the logical reasoning on the behavior of parallel programs.

Additionally, we will see how visualization in sequential, parallel, and constraint programs help in verifying our understanding of the *global picture*, and it is a valuable tool in order to identify inefficiency points. If necessary, the display can be used to guide a program restructuring (e.g., a different task partitioning or a change in parallelization conditions) which increases program efficiency.

2.10 Concurrency in Logic Programming

Concurrency has been studied in the context of many programming paradigms, and different mechanisms to express concurrent computations have been devised in the realm of procedural and object oriented languages [BA82, Han77, And91, AS89].

Concurrent logic languages usually refers to those which implement a sort of committed choice semantics, but some other languages which feature encapsulated execution with deep guards which may promote and cause non-deterministic reductions can also be termed as concurrent.[10] These are not, however, the two only possibilities: we will see that our proposal belongs to an altogether different case.

2.10.1 Concurrent Logic Languages

Languages of the first group are named as a whole concurrent logic languages. This group includes Parlog [CG86], Concurrent Prolog [TSS87], Guarded Horn

[10]This classification is perhaps not an usual one, but in [Tic95] a similar taxonomy is made, albeit with a different terminology. [Tic95] examines expressiveness and implementation techniques of the former, but not of the latter.

Clauses [Ued87], Janus [Sar90] and others [Sha89, Tic95]. These share a number of characteristics. First, concurrency is implicit, i.e., a clause literal represents a concurrent process. Even if this is an elegant interpretation of Horn Clauses at first sight, it can cause the generation of an unnecessarily large number of concurrent processes, and impedes writing sequential code. It seems more advantageous, in general, generating concurrent computations explicitly by means of language primitives (or, conversely, that the language have operators to express sequential composition of processes) [HC94, CH96]. Although there is still debate in this sense, it is remarkable that some languages which started as implicitly concurrent, such as Oz [Har96] have finally opted for explicit concurrency in more recent versions [HF00].

Another common characteristic of concurrent logic languages is that communication and synchronization is performed through shared variables, which represent a channel in which communication happens when variables are instantiated. Synchronization happens when processes wait for some variables to be instantiated (sometimes to a given value). This very attractive model has various practical problems. The first and more important is that backtracking is highly difficult to implement (and, according to a reactive view of the computation, inadequate). Thus, these languages opted for removing backtracking altogether. We think, however, that backtracking is part of the control model of Prolog, and that eliminating it is, simply, an unacceptable option.

2.10.2 Deep Guards and Non-Deterministic Concurrent Languages

Although probably not conceived as concurrent logic languages, a more modern family of logic languages, including Andorra-I [SCWY91a, War88] and AKL [JH91] tackle the problem of backtracking in concurrent languages while still maintain variable sharing among different goals. Due to the difficulty of mimicking the sequential backtracking behavior, they adopt an alternative selection rule, or an *encapsulated* backtracking.

In Andorra-I the computation performed by every process is suspended unless a deterministic path can be followed; if that is not possible, computation is performed sequentially. Although Andorra-I is most frequently perceived (and not incorrectly) as a parallel language with a well-defined, alternative execution rules, suspension of execution agents depends on the values of the arguments, and not only on processor availability. This point makes Andorra-I a concurrent language, as process evolution depends on explicit program data.

AKL allows encapsulated search while, at the same time, deterministic bindings are communicated outside the encapsulated computations. Although this solves some of the problems associated with backtracking, it still has some drawbacks: apart from supporting an explicit concurrency approach, they require very specialized implementation technology, particularly in what respects variable representation, which in some cases leads to a poorer performance with respect to sequential execution. Besides, the operational semantics of these languages (specially that of AKL) if far from Prolog's.

2.10.3 Reusing Existing Machinery

Another interesting focus toward concurrency is using the existing capabilities of parallel Prolog implementations, such as *&–Prolog* [HG91], Aurora [Lus90], MUSE [AK90b] and ACE [GHPSC94]. These systems proved time ago the possibility of creating resilient and efficient multiprocessor Prolog implementations. In fact, some of them have been used to build (parts of) concurrent applications (Aurora in [SMS96]) or even concurrent and distributed Prolog systems [CH96, HCC95]. From the point of view of internal machinery, although these systems were designed as WAM extensions, we feel that for a practical purpose, a model which requires less modifications for the WAM is useful, and we are interested in better solutions for communication and synchronization among execution threads.

2.10.4 Other Proposals of Concurrency and Communication

The difficulties associated to the variable-based communication lead to the development of other means of communication and synchronization. One of them is that of ports [JH91], used, e.g., in MT-SICStus [EC98].

MT-SICStus has a relatively simple design, with some details taken from Erlang [AVWW96]. Threads can be created (with an initial goal) and removed. Goal copies can be sent and received using a port in each thread. This allows creating simple goal servers which execute any goal received. This model is certainly useful, but it lacks a clean interaction with backtracking (what is send through a port is not recovered on backtracking), and the aforementioned goal server interacts with the module system. Besides, coding explicitly that goal server seems a must in most applications.

Threads are started by means of an explicit construct in Oz 2.0 [Har96], which returns a value as if it were an expression evaluation. Message passing and synchronization use shared variables and an abstract data type, the *Port*, which can be shared among several processes and handed down to functions and procedures. A shared data zone allows fast communication among threads. A very interesting characteristic of Oz is the coherence and caching mechanism used when performing concurrent and distributed executions.

Erlang [AVWW96] is a functional language whose ideas impregnate MT-SICStus. Synchronization and communication are made by using a single *port* in every thread. Waiting on a port causes suspension until an atom arrives.

An alternative to ports is using Linda blackboards [CG89b, CG89a]. Linda is a simple but powerful communication paradigm which focuses on (and unifies) concurrency and synchronization mechanisms. The Linda model assumes a shared data area (the blackboard) where tuples are stored and retrieved from, using pattern matching, by concurrent processes. The read and write operations are made by atomic primitives. This mechanism has been provided, either by means of native support or as libraries, in different logic programming systems [BC91, Swe95, CH96, HCC95, Tar99, De 89].

A practical example of this approach is the Jinni [Tar99] system, based on Bin Prolog. Jinni has a relatively rich set of primitives to create threads, giving them an initial goal to work on, and to recover the solutions to a goal until the call finitely fails. Coordination among *engines* (the entities which perform work in Jinni) happens through a set of primitives similar to that of Linda, which access and modify a shared blackboard. These operations allow reading and writing (using pattern matching) to and from the blackboard, and collecting all the tuples which match a pattern. The synchronization can be based on constraints, and it gives the possibility of migrating computations to other hosts. Jinni is attractive, but in our opinion, the shared blackboard is a device external to the Prolog model and language, and it is not really needed.

2.10.5 Proposals Based on a Global State

Other proposals [BC91, Sha86] fall into the category of parallel languages with a global environment, and need an explicit control in the interactions. Our approach belongs to this last category. The main difference with a synchronization based on Linda or in another form of blackboards, is that in the former case the programmer is forced to use the interface provided (probably by means of a library) to an external entity. Our approach incorporates some characteristics related to a global blackboard, increasing Prolog's semantics when necessary.

Chapter 3

Visualization of Sequential, Constraint, and Parallel Logical Programs

Chapter Summary

This chapter addresses the design and implementation of visual paradigms for observing the execution of logic programs. Three different approaches, tailored to three different needs, are discussed: visualization of sequential execution, visualization of parallel execution, and visualization of constraints.
Sequential program visualization focuses on depicting the search tree generated by a query to a program, and includes the capability of looking at the actual values of the parameters at run time. Parallel program visualization focuses more on control-related issues, reflecting very faithfully the time characteristics of parallel executions. The last visualization schema shown targets the display of data in CLP executions, where representation for constrained variables and for the constrains themselves are seeked. In all three cases, abstractions of the proposed depictions are discussed. Tools exemplifying the devised depictions have been implemented, and are used to showcase the usefulness of the visualizations developed.

3.1 Introduction

Program visualization has been classically used by computer scientists for many different purposes. We are not referring here to the use of programs to represent graphically data or processes (undoubtedly of outmost interest), but rather to the use of visualization to depict programs, either statically (recall, for example, the flowcharts or block diagrams) or dynamically; in the last case, both control and data evolution can be represented.

Visualization of Prolog (and, in general, CLP) executions is receiving much attention recently, since it appears that classical visualizations are often too dependent on the programming paradigms they were devised for, and do not adapt well to the nature of the computations performed by CLP programs. Also, the needs of CLP programmers are quite different [Fab97] from those of programmers working with more traditional paradigms. It is not that the types of data objects used in CLP languages (e.g., lists, n-ary trees...) cannot be implemented in procedural or object-oriented languages, but rather that they are *primitive* objects of the target languages, and as such we expect them to be directly represented by a visualization tool. In a language where those types of data have to be explicitly expressed and handled, we might delegate the task of visualizing them to specialized procedures.

Basic applications of visualization in the context of CLP, as well as Logic Programming (LP), include[1]:

- Debugging. In this case it is often crucial that the programmer obtain a clear view of the program *state* (including, if possible, the program point) from the picture displayed. In this application, visualization is clearly complementary to other methods such as assertions [AM94, DNTM89, BDD$^+$97] or text-based debugging [Byr80, Duc92b, Fer94, EJ99]). In fact, many proposed visualizations designed for debugging purposes can be seen as a graphical front-end to text-based debuggers [DN94].

- Tuning and optimizing programs and programming systems (which may be termed—and we will refer to it with this name—as *performance debugging*). This is an application where visualization can have a major impact, possibly in combination with other well-established methods as, for example, profiling statistics.

- Teaching and education. Some applications to this end have already been developed and tested, using different approaches (see, for example, [EB88, Kah96]).

In all of the above situations, a good pictorial representation is fundamental for achieving a useful visualization. Thus, it is important to devise representations that are well suited to the characteristics of CLP data and control. In addition, a recurring problem in the graphical representations of even medium-sized executions is the huge amount of information that is usually available to represent. To cope successfully with these undoubtedly relevant cases, *abstractions* of the representations are also needed.

[1] These applications are, of course, not exclusive of (C)LP.

Here, “abstracting” refers to a process which allows a user to focus on interesting properties of the data available, by filtering pieces of unnecessary information and which will hinder the comprehension of the phenomena under study rather than easing it. Ideally, such abstractions should show the most interesting characteristics (according to the particular objectives of the visualization process, which may be different in each case), without cluttering the display with unneeded details.

The aim of the visualization paradigms we discuss is quite broad—i.e., we are not committing exclusively to teaching, or to debugging—, but our focus is debugging for correctness and, mainly, for performance. This focus coalesces totally with our quest for more efficiency in the execution of logic programs: the tool and techniques presented in this chapter can be seen as an aid to explore and understand the behavior of logic programs. But, interestingly, the dual view can also be taken: the performance of a logic programming system can be evaluated (and this is, arguably, the only relevant factor to an end user) with respect to programs being executed. Inspecting how programs with known characteristics execute gives information on how the underlying machine performs, and thus it helps to uncover weaknesses.

In the next sections we will present an approach to the study of the run–time behavior of logic systems based on devising *visualization paradigms* which represent the execution of such systems. The gap between the general characteristics of program execution in the system under study and the final visualization paradigm used to represent it will be filled using a methodology based on stepwise refinement. This refinement leads to a number of different visual representations which will be devised in a natural fashion for a number of models of execution in logic programs, based on the structure of the dependencies that hold for the execution represented.

The visualization paradigms we will deal with can be divided into three non-exclusive categories: visualizing the execution flow / control of the program, visualizing the actual variables (i.e., representing their run-time values), and visualizing constraints among variables. The last two categories are specially difficult to handle in CLP, and do not have a clear counterpart in procedural languages—at least, not at the level of being “first class” citizens of the language.

In the following sections we will give an account of three approaches developed:

- Visualization of sequential program execution (starting at Section 3.3), paying attention to issues related to creation and binding of variables (either LP or CLP), and

- Visualization of parallel program execution (starting at Section 3.8), focusing on control and resource (e.g., time, processor, memory) usage, and

- Visualization of the history of the data in a program (starting at Section 3.11), focusing on the display of CLP variables and the relationships (i.e., constraints) among them.

These three approaches can be used for any of the applications mentioned previously. Obviously they can be used to teach, since an intuitive view of the execution is displayed, giving sometimes the possibility of replaying an execution and showing how the rules of the language led the control flow. They can also be used to debug user programs, since the views shown by the tools come from actual executions. Each tool offers a different representation of the execution, and thus it can be used to locate different types of program errors or performance problems.

3.2 Basic Notions and Methodology

In order to derive homogeneous visualization paradigms starting from the basic properties of different execution models we propose the use of a methodology loosely based on stepwise refinement.

We briefly introduce three basic notions for this purpose (since these concepts are quite primitive, we will rely somewhat on the reader's intuition to avoid verbosity): An **observable** is any generic characteristic of the system under study whose variation we want to track. An **event** is a uniquely distinguishable instantiation of an observable in an execution. A **trace** is a collection of events which corresponds to a particular execution. Observables abstract out details of concrete operations, and allow concentrating on characteristics perceived as interesting for study. Events usually include their type, which names the corresponding observable, and some additional information, which distinguishes a particular event from other events of the same type. Such information may include for example time stamps, invocation number, goal and agent[2] identifiers, etc. In general it is required that no two identical events can ever appear in the same execution—the use of unique time stamps can ensure this. Traces can gathered using a variety of methods (e.g., instrumenting a real system, or by means of simulation), and can be stored in trace files or be interactively obtained as the visualization proceeds. Each method has its advantages and drawbacks.

[2]Throughout the paper we use the term "agent" (or worker) to refer to the process, normally mapped on an agent, that is working on a task.

Once an execution model is chosen the first step is to decide at what abstraction level the model is to be visualized. This is done by defining the observables and the information which will be encoded in the events[3]. Also, a notion of dependency among events is defined. A graph structure results from this dependency relation which is then used as the basis for developing the visualization paradigm. Finally, common characteristics of the possible graph structures generated should be identified and unifying principles found in order to devise a visualization paradigm. Such a paradigm should be as simple, flexible, accurate, and intuitive as possible, reflect the structure of the graph, and hopefully be homogeneous across execution models and types of parallelism. Of course, the final result can certainly only be satisfactory if the resulting visualizations prove to be useful enough in practice for program development or system debugging, and this may require several iterations through the paradigm design cycle.

Finally, some models can be seen as a combination of several individual models. A visualization paradigm can often be derived in such cases by combining the corresponding individual visualization paradigms in a way that mimics the combined execution model. Sometimes this is not possible (properties of independence do not hold, or the graphical representation is not intuitive or elegant). In these cases the problem has to be tackled from scratch as a whole, and perhaps even different observables will have to be defined in order to arrive at a satisfactory representation.

3.3 Sequential Search Tree Visualization

One of the main characteristics of declarative programming is the absence of explicit control. Although this theoretical property results in many advantages regarding, for example, program analysis and transformation, programs are executed in practice with fixed evaluation rules,[4] and different declaratively correct programs aimed at the same task can show wide differences in efficiency (including termination, which obviously affects total correctness). These differences are often related to the evaluation order. Understanding those evaluation rules is important in order to write efficient programs. In logic languages where these rules are complex (for example, the Andorra family [War88, FHJ91, War93] or, as we will see later, concurrent or parallel programs in which the scheduling can be very different among runs) a high level vision of the execution is of great help. In this context, a good visualization of the program execution (probably

[3]Note, also, that a visualization tool (or any other tool) can change the abstraction level by ignoring the some of the information contained in the events.

[4]By "fixed" we mean that these rules can be deterministically known at run-time, although maybe not known statically.

Observable	**Comment**
NEW_CALL	A new call is performed
TRY_HEAD	The next untried clause is tried
HEAD_SUCCESS	The tried head finished unified with the call
HEAD_FAILURE	The tried head did not unify
CALL_SUCCESS	All goals in the current clause succeeded
CALL_FAILURE	No clause for the goal succeeded

Table 3.1: Common observables for sequential execution of logic programs

combined with other tools) can help to uncover performance (or even correctness) bugs, either in the user code or in the system programmer code.

3.3.1 Choosing Observables

Sequential execution display can draw its graphical representation directly from the resolution trees traditionally used in logic; this does not mean that this representation is the best suited for any purpose, but in any case it may serve as a basis for future developments. Additionally the representation can be adorned to include information pertaining characteristics not included in the raw depiction of the search tree.

The *observables* we want to register have a strong operational flavor, even if we want to display a resolution tree. This so because we are interested in displaying in a performance-oriented fashion, and therefore we are interested in retaining most of the operational characteristics. Apart from the global success or failure of a (toplevel) goal (which will possibly give us a too coarse view of the execution, but which can however be used to abstract parts of the execution—see Section 3.7), the fundamental observables are the invocations, backtrackings, successes, and failures of goal calls. The observables we have chosen are shown in Table 3.1; they resemble the *ports* used by the debuggers based on the *Byrd Box* model [Byr80], although they are slightly more fine-grained. This level of detail is motivated by the desire of displaying failed head unifications, which some Byrd-box based debuggers do not show. As an example, the code

```
q(a,a).
q(b,b).
q(c,c).
```

traced using a industry-standard Prolog system produces the following output:

```
| ?- q(X, b).
        1      1 Call: q(_158,b) ?
        1      1 Exit: q(b,b) ?
```

where the failure of the unification of the first clause head is absent. This makes it difficult to map immediately the trace to the textual program source, and makes correctness debugging more involved: we are missing failing clauses in the trace, and these clauses might be the ones leading us to a solution. Hence the finer granularity of our observables.

The sequential order of generation of observables gives by itself a natural way of stamping (and identifying) events and producing unique instantiation of observables. This ordering information is however not enough to render a tree: knowledge about the relationship of a given event with (a subset of) the preceding events is needed in order to construct the search tree. As an example, all events should specify which part of the tree they correspond to, either by giving absolute coordinates in the tree (e.g., by coding the path from the root) or by providing identifiers of parent/sibling events.

There are some event properties which can be used to derive a depiction: for example, TRY_HEADs should always refer to a previous, already existing NEW_CALL (ditto for HEAD_SUCCESSes, HEAD_FAILUREs, CALL_SUCCESSes, and CALL_FAILUREs), and alternative clauses for a call always use the same runtime arguments. Adding this information to the events will allow us to relate subsequent observables with a given initial invocation; this, as we will see, makes it easy to represent dynamically the evolution of the execution. Since all the events in an execution can be related to NEW_CALLs, only unique identifiers for NEW_CALLs are needed.

Additionally, there are some ordering constraints which must be met by the events: no TRY_HEAD can appear unless a previous NEW_CALL with same identifier was emited before; no HEAD_SUCCESS or HEAD_FAILURE can appear unless a previous TRY_HEAD event referring to the same call (i.e., with the same identifier) was issued, and they must be balanced; only a CALL_FAILURE for every NEW_CALL must appear; etc. These per-invocation constraints on events are summarized as a DFA in Figure 3.1. TRY_HEAD$_i$ refers to the i^{th} clause of the predicate being called; every time a new clause is tried, i is incremented until the of clauses of the predicate are exahusted—or a cut is executed in a clause.

These observables do not take into account how unifications take place in each call, although we may have included explicit events for that purpose. In the latter case, HEAD_SUCCESSes and HEAD_FAILUREs should have been decomposed on unifications.[5] Probably showing by default information about runtime arguments in the search tree will

[5] Note that it is possible to approximate the latter behavior by decomposing at source level the head unification into explicit unifications in the body of each clause, and then using the shown events.

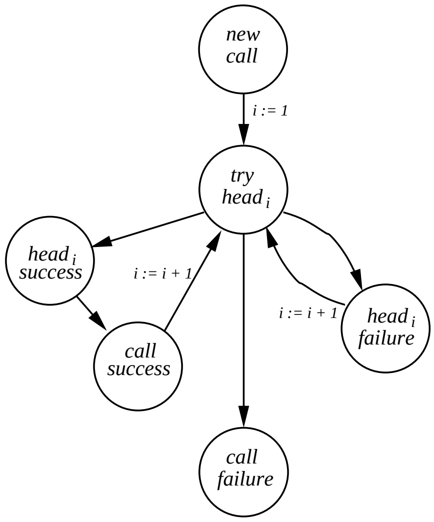


Figure 3.1: Precedence in the events for sequential tree display

provide too much detail; we will, at the moment, focus on showing the control-related part of the execution, and leave the data display for a later moment (and, possibly, under user request). Thus we do not need to include explicit events related to argument unification at this level.

3.3.2 Choosing a Depiction

From the initial description and dependencies, we can derive a graph which represents the execution of a program: NEW_CALLs are represented by nodes in the graph (one per invocation), and the events related to a particular invocation are dynamically reflected in the graph by adorning it with labels (at least, at the moment). The structure of the execution is depicted by using *caller/callee* dependencies. Note that if we want to fully represent how the search process took place (which will help us to have a more useful tool, see Section 3.5.2), we need to distinguish the different subtrees arising from different clauses, i.e., we need to distinguish And continuations from Or alternatives. One way to perform this is, keeping the idea of having a node per invocation, to label the edges of the graph so that it is apparent which correspond to an alternative and which correspond to a continuation.

To illustrate this, we will use the program in Figure 3.2 with the query `?- a(1, 2).` The graphs in Figure 3.3 represent, from left to right and from top to bottom, several stages in the execution of the query. The nodes are labeled with the name of the predicate being called, and adorned with the state of that node. Throught the rest of the section

```
a(X,Y):- b(X,Z), c(Z,Y).
a(X,Y):- c(X,Y).

b(1,2):- true.

c(1,2):- true.
```

Figure 3.2: Sample program

we suppose that the text of the program is available when displaying the execution (it could, if necessary, have been emitted with the trace), so that any information pertaining to the program source is available. We have used a lighter color to represent *continuation* (sibling) edges and a darker color for edges representing a father/children relationship. Thus, a series of nodes linked with light-colored edges correspond to the *body* of a clause, and they appear in the order in which they were executed (i.e., textual order, for standard Prolog). For the different alternative clauses of a predicate, we rely on the order in the graphical depiction. Although at any given moment only one clause is responsible forthe success of a clause, it is important to detail the order in which clauses were tried; otherwise, the result of side-effect and control-related predicates could not be portrayed properly.

In Figure 3.3, from left to right, the toplevel goal `a/2` is called first. Its first clause is tried, and the head unification also succeeds. At that point the clause is "expanded", and the calls to `b/2` and `c/2` have to be performed. The first clause of `b/2` is tried, and it suceeds. The first clause of `c/2` is then tried, and the head unificacion fails; since there is only one clause of `c/2` the whole call fails. A new clause is then tried for `b/2`, and the call fails. The first clause for `a/2` has then failed, and the second clause is tried with a new TRY_HEAD. This clause calls `c/2` and it suceeds. Then, in the same graph, we have the whole search tree with Or alternatives from top to bottom, and And continuations from left to right.

Since the graph reflects the state of the execution at any given time, this depiction shows a quite faithful history of the execution, specially regarding the structure of the recursive calls. This can be used to explore and understand, for example, where the program made most of the calls, which parts of the program fail the more often, whether there are frequent left recursions (which, in a system with last call optimization, uses up comparatively more memory), and other characteristics which affect performance. As it is not static, but updated as the program proceeds, the user has, at least in principle, the possibility of stopping the replay and examining the state of the computation up to

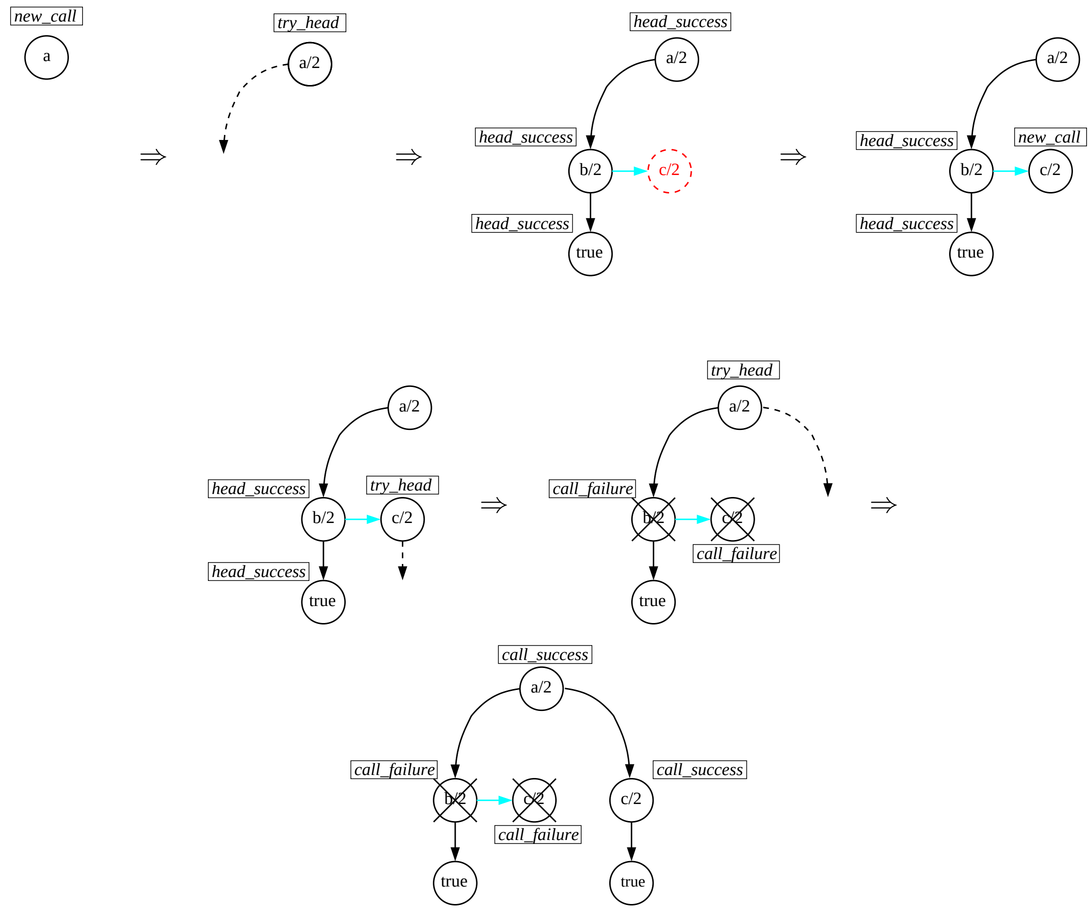


Figure 3.3: Successive graphs for a sequential execution

the point allowed by the information contained in the events. On the other hand, the graph display, although having many of the characteristics we want to show in a program depiction, is still probably overloaded with gaudy decoration which might obscure rather than enlighten the structure of the execution. For this, we will, based on these graphs, develop a more stylized representation.

A Note on Alternative Search Rules: The timestamp suggested for the events is not only a convenient way of generating unique identifiers: it is also of utmost importance to tackle the representation of alternative search rules. The order in which calls were executed in a given search tree is ultimately defined by the timestamps. As an important example which we will discuss more in depth in Section 3.8 and following, parallel execution can be seen as an alternative execution rule, which is only known at run-time, and which depends on external facts such as the load and scheduling characteristics of the machine.

3.4 AORTA Trees and the *APT* Tool

In order to test the basic ideas of the previous section, we designed and implemented a tool named *APT* (*A Prolog Tracer* [Lue97]) which serves as a control visualizer for CLP languages. *APT* is essentially a search tree visualizer based on the TPM [EB88],[6] and inherits many of its characteristics. However, *APT* also adds some interesting new features. *APT* is built around a meta-interpreter coded in Prolog which rewrites the source program and runs it, gathering information about the goals executed and the state of the store at run-time. This execution can be performed depth-first or breadth-first, and can be replayed at will, using the collected information. All *APT* windows are animated, and are updated as the (re)execution of the program proceeds.

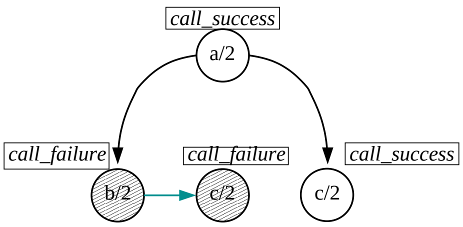


Figure 3.4: Graph for an execution of the program in Figure 3.2

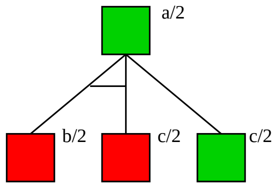


Figure 3.5: AORTA execution tree for the program in Figure 3.2

The main visualization window of *APT* provides a tree-like depiction, derived from that shown in the previous section. Nodes in this portrayal represent invocations, and the different events are shown with color codes which correspond to the *state* of the invocation (white when the call corresponding to a node has not yet been performed, yellow when the node is CALLed or reTRIed, green for SUCCESSed and red for FAILed). Additionally, calls are adorned (optionally) with the name of the predicate being called. Figures 3.5 and 3.4 show the execution graph and AORTA tree corresponding to this program with the query `?- a(1, 2).`. The figures show all the search process, including failed paths, so it is easy to perceive, at a glance, all the work performed until a solution is reached. Note that these figures do not show the call corresponding to the (implicit) `true` in the facts. This is more adequate for real programs, and avoids having too many objects in the display.

There has been a graphical transformation from the graph to the AORTA tree: the *continuation* edges in the graph, linking nodes in execution order, are replaced by a line

[6]Another CLP visualizer that depicts the control part as a tree in the TPM spirit is the one developed by PrologIA [Pro98].

grouping the goals inside each clause, and all of these nodes are linked to the parent call. The order is now implicitly represented by the left-to-right order of nodes in the portray. Thus, goals in the body of the first clause of `a/2` (`b(X, Z)` and `c(Z,Y)`) are shown as nodes whose edges to their parent are crossed with a line—these are And-branches corresponding to the goals inside the clause. The goals in the body of the second clause (only one, in this case) are linked to the parent node with a "separate" set of edges. The actual run-time arguments are not shown at this level, but nodes can be blown up for more detail, as we will see later.

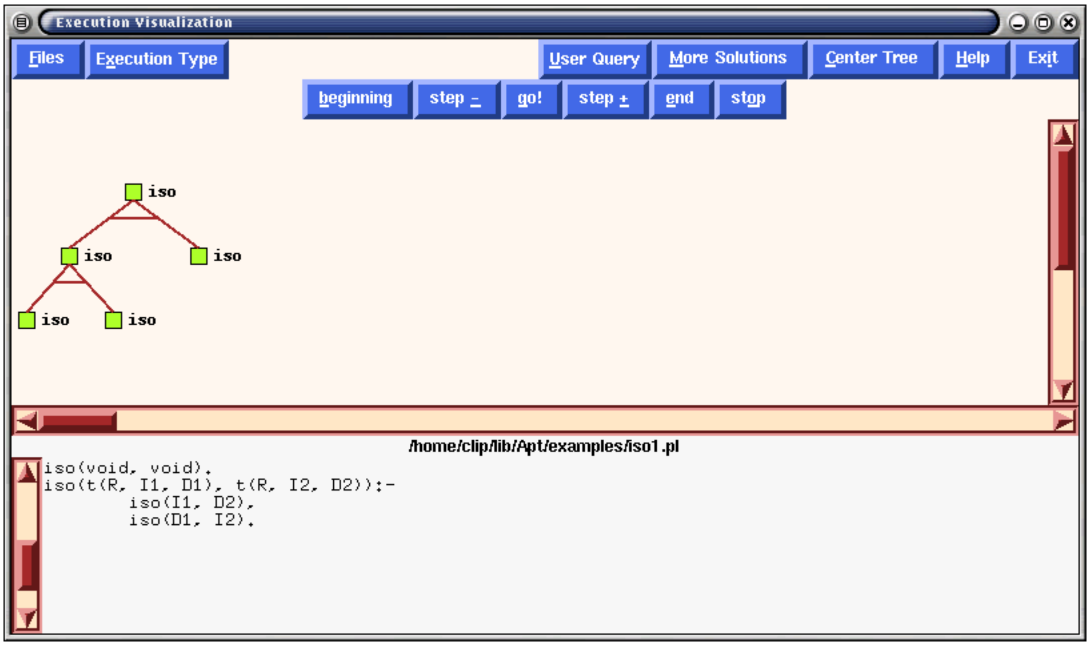
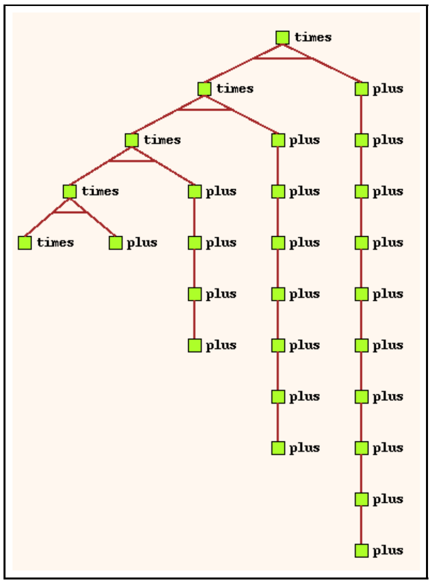


Figure 3.6: A small execution tree, as shown by *APT*

3.4.1 Control in *APT*

Figure 3.6 shows a view of an actual *APT* window. The buttons at the top are used to load programs, execute queries, and navigate through the execution. The bottom frame is a full-featured text editor, and the top frame displays the execution tree. In this representation, calls to user code are drawn as squares and nodes corresponding to built-ins appear as circles. The portray of an execution tree gives information on the amount of work performed in the execution. As an example, Figure 3.7 shows the execution tree and code for a simple version of the `times/3` predicate using Peano arithmetic [SS87]. This predicate is called as `times(M, N, R)` (`M` = 4, `N` = 3 in the example). It is easy to see that most of the work is performed on the right side of the tree and that it is the `plus/3` predicate the one which performs this work. Moreover, it is as well easy to perceive a pattern: `plus/3` is executed a different number of times in each execution step, which suggests[7] a complexity of, at least, quadratic on some argument (this the rate at which the series 1+ 4 + 7 + 10..., the number of calls to `plus/3`, grows), and linear in some other (because the increment of `plus/3` in different recursion steps is not fixed). This reasoning might not be apparent from this example only, but trying a couple of different cases would help to clarify what each argument is responsible for in the picture. The complexity is actually $O(M^2 \cdot N)$, which can be straightforwardly derived by mapping the search tree in the picture to the actual code: it is easy to spot that the running times of `plus/3` is governed by its first argument, and this argument is bigger in the outer recursion steps of `times/3`, since it is the result of a series of previous multiplications. Moreover, the predicate is right-recursive, which is also also easy to deduce from the tree.



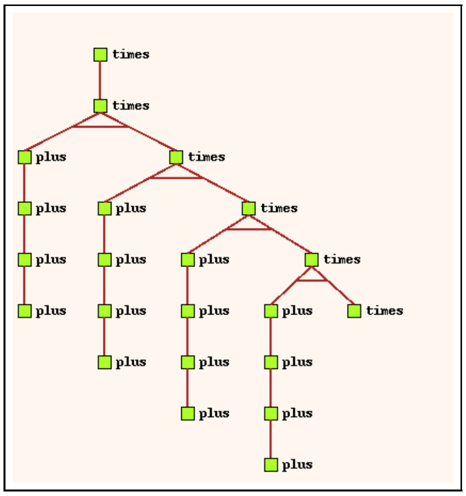

```
example(R):-
        times(s(s(s(s(0)))), s(s(s(0))), R).

plus(0,X,X).
plus(s(X),Y,s(Z)):- plus(X,Y,Z).

times(0,_Y,0).
times(s(X),Y,Z):-
        times(X,Y,Z1),
        plus(Z1,Y,Z).
```

Figure 3.7: Execution tree and code for a left-recursive, non-accumulative multiplication

The result of removing left-recursion (an accumulating parameter will suffice in this case) is shown in the search tree and code in Figure 3.8. The layout of the tree generated by the same call is notoriusly different: it clearly right-recursive, the bulk of the work being performed by the `plus/3` predicate on the left part of the tree. Also, the amount of work is clearly smaller: the number of calls to `plus/3` is constant for every recursion step of `times/3`, and the leaning rectangle-shaped pattern suggests a $O(M \cdot N)$ complexity.

[7]Of course a complexity study will reveal the exact order; but we are, precisely, trying to give an intuitive understanding of the behavior of the programs being examined.

Looking at the code it is clear that the size of the calls to `plus/3` depends solely on the second argument of `times/3`, which does not change in different recursive calls.




```
example(R):-
        times(s(s(s(s(0)))), s(s(s(0))), R).

plus(0,X,X).
plus(s(X),Y,s(Z)):- plus(X,Y,Z).

times(X, Y, Z):- times(X, Y, 0, Z).

times(0, _Y, Z, Z).
times(s(X), Y, Accum, Z):-
        plus(Y, Accum, Mid),
        times(X, Y, Mid, Z).
```

Figure 3.8: Execution tree and code for a right-recursive, accumulative multiplication

Even in cases where left- and right recursion do not lead to different complexity results, using right recursion is usually better; in many systems implementing last call optimization, right recursive predicates usually run in constant memory space, since the stack space holding local variables can be released prior to the last call. This case is also easy to detect (and understand) using search tree depiction. We will use an example (code omitted, due to its simplicity) which adds the numbers from `1` to `N`, using a non-tail-recursive program (Figure 3.9) and a tail-recursive program (Figure 3.10). The amount of work is almost the same in both cases (the number of nodes in the tree can just be counted, or, more roughly, estimated at a glance), but most Prolog implementations would favor the tail-recursive implementation. In Figure 3.9 the branches on the right of tree are the operations for which local variables have to be presumably kept in the local environment. These branches do not appear in Figure 3.10; albeit the latter has an arguably less pleasant aesthetics, the use of environment space disallows very important optimizations in the memory usage.

Execution can be performed either in depth-first or breadth-first mode. In the case of depth-first search, the user can specify a maximum depth to search; when this depth has been reached, the user is warned and prompted to decide whether to stop executing, or to search with a new, deeper maximum level. The search mode and search depth are controlled by the metainterpreter built in *APT*, so that no special characteristics are

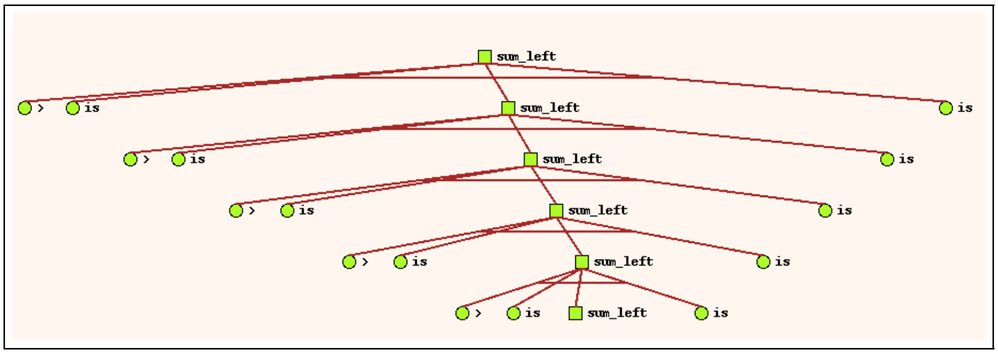


Figure 3.9: Non tail-recursive simple predicate

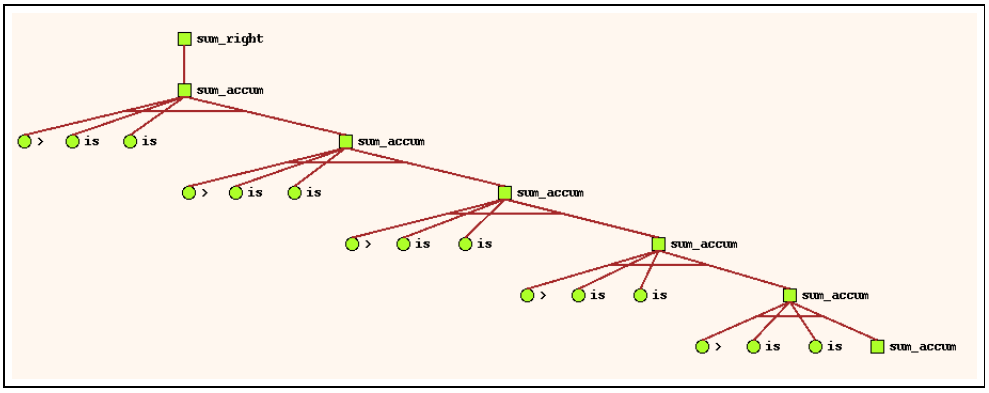


Figure 3.10: Tail-recursive simple predicate

required from the underlying CLP system. *APT* treats correctly the cuts in program, showing the pruned alternatives; it also deals properly with the metacalls, since the whole program is itself metaexecuted.

3.4.2 Visualizing Data in *APT*

Clicking on a node of the tree opens a different window in which the relevant part of the program source, i.e., the calling body atom and the matching clause head, are represented together with the (run-time) state of the variables in that node, including the input/output modes. The default depiction offered by *APT*corresponds to the Herbrand domain. This depiction is is built into *APT*, and it was chosen since that most CLP systems include the Herbrand domain as default, in order to be able to build data structures. The following examples will focus on the visualization of Herbrand terms (although alternative representations are possible, see Section 3.11.2) and we will veer to constrained variables later in the chapter.

As a simple example, Figure 3.11 shows a blow-up of a leaf node of the execution in Figure 3.6. The run-time call appears on top at the right, and the matching head is placed below it. The answer substitution (i.e., the result of head unification and/or execution of the clauses in the body) is shown enclosed by rounded rectangles. The arrows represent the source and target of the substitution, i.e., the data flow: variable $\texttt{I2}_1$ was free at the time of the call, and is unified with the `void` constant. On the other hand, $\texttt{D1}_1$ was already bound, and matches the `void` constant in the source.

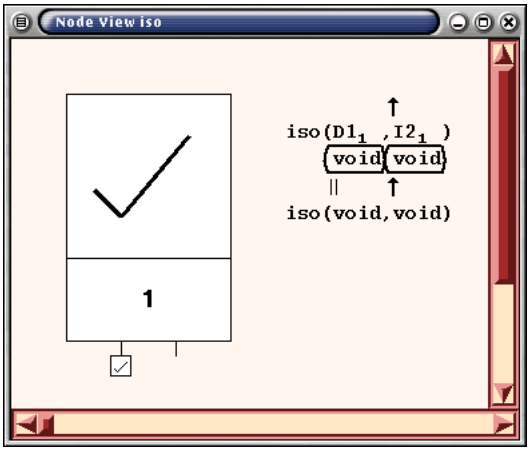


Figure 3.11: A simple leaf node

The diagram on the left of the program text (still in Figure 3.11) represents the state of the call: marked with a tick ($\surd$) for success (as in this case), crossed ($\times$) for failure, and signaled with a question mark if the call (either first call or subsequent backtracking) has not finished yet. The number below the symbol denotes the number of clauses in the predicate. For each of these clauses there is a small segment sticking out from the bottom of the box. Clauses tried and failed have a "bottom" ($\perp$) sign; the clause (if any) currently under execution, but not yet finished, has a small box with a question mark; and a clause finished with success is marked with a tick. In this case the first clause suceeded.

In the example shown in Figure 3.12 the arguments were already instatiated to structures with variables inside. Some of these variables receive values from other goals inside the clause (variables $\texttt{I1}_1$, $\texttt{I2}_1$, and $\texttt{D2}_1$ in the head, which unify with $\texttt{T1}_0$, $\texttt{T2}_0$, and $\texttt{X}_0$), and others receive their bindings due to variable sharing in the clause head: variable $\texttt{Val}_0$ unifies with $\texttt{R}_1$, which appears twice in the clause head, and it subsequently *receives* a 2 from the call.

The presentation of these node views depends on the type of data (i.e., the constraint domain) used. This is one of the most useful general concepts underlying the design of

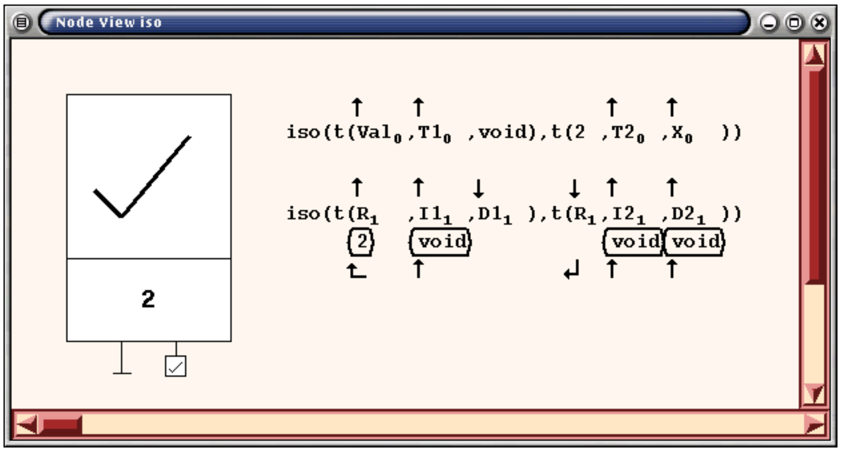


Figure 3.12: An internal node with different instantiation modes

APT: the graphical display of control is logically separated from that of data. This allows developing data visualizations independently from the control visualization, and using them together. The data visualization can then be taken care of by a variety of tools, depending on the data to be visualized. Following the proposal outlined in the previous section, this allows using *APT* as a control skeleton for visualizing CLP execution. In this case, the windows which are opened when clicking on the tree nodes offer views of the constraint store in the state represented by the selected node. These views vary depending on the constraint domain used, or even for the same domain, depending on the data visualization paradigm used (e.g., if different visualizations for the same type of data are chosen). Visualization for CLP data will be discussed more in deph later in this chapter.

A prominent feature of *APT* is its ability to show the origin of the instantiation of any variable at any moment in the execution. In order to do that, *APT* keeps track of the point in the tree in which the (current) substitution of a variable was generated. Clicking on a substitution causes a line in the main tree to be drawn from the current node to the node where the substitution was generated. Figure 3.13 shows a part of an execution (from a 5-queens problem) in which a black line from a `no_attack` node points to an invocation of `queens` as the instantiation origin of a variable. This is a very powerful (and expensive) feature which helps in correctness debugging, as the source of a (presumably) wrong instantiation (causing, for example an unexpected failure or a wrong answer) can be easily located. The culprit node can in turn be blown up and inspected to find out the cause of the generation of those values.

A similar capability can be found in the *Color Prolog* tool [NKD97], where the "his-

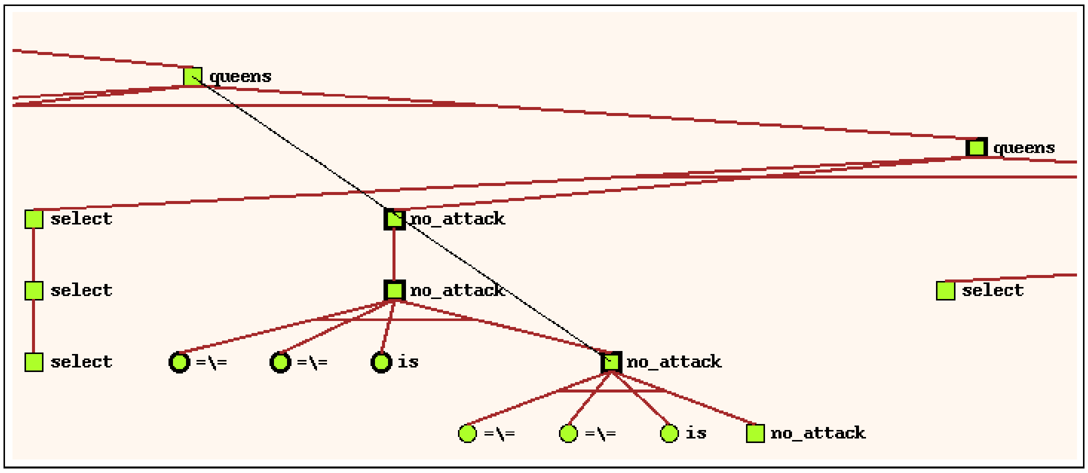


Figure 3.13: Showing the origin of an instantiation

tory" of every variable in the execution is depicted using a different color. Variables (or, in general, terms) describe thus a colored path in the search tree from the moment in which they are created up to the leaves. This tool is undoubtedly of much value for teaching (which is its original aim), but it probably lacks some abstraction properties and the performance debugging orientation we pursue. Quite interestingly, the authors had to face the issue of color unification.

3.4.3 Some Implementation Details

The tool reads source programs, "enriches" them (basically numbering each clause and marking variables so that tracing their origin is easier and faster), and metaexecutes them, producing an internal trace with information about the search tree, the variables in each call, and the run-time (Herbrand) constraints associated. *APT* uses a "rich meta-interpreter", in the sense that it keeps track of a large amount of information, and executes an "enriched program": the program is rewritten so that it helps the metainterpreter. In retrospect, the "rich meta-interpreter" approach has advantages and disadvantages. On one hand it allows determining very interesting information such as, e.g., the origin of a given binding. On the other hand, it cannot cope easily with large executions, both due to time and memory consumption.

After reading and running the program, the execution can be replayed, either automatically or step-by-step, and the user can move forwards and backwards. More detailed information about each invocation can be requested by accessing the node with represents the call. The tool has a built-in text editor, with a full range of editing commands, most of them compatible with the Emacs editor, and files open from the visualizer are loaded into the editor. The queries to the program are entered in the window *APT* was launched from. Once a query has been finished, the user can ask for another solution to the same query. As in a toplevel, this is performed by forcing backtracking after a simulated failure. If the tree to be visualized is too large to fit in the window (which is often the case), slide bars make it possible to navigate through the execution.

APT uses Tcl/Tk [Ous94] to provide the graphical interface. The original implementation of *APT* was developed under SICStus Prolog. It has also been ported to the clp(fd)/Calypso system developed at INRIA [DC93].

3.5 Programmed Search and the Enumeration Process

The control-focused observables proposed so far are not only useful for traditional logic programming, but also for constraint logic programming, since the control behavior of most CLP systems is inherited from that of Prolog. Regarding control, in CLP programs (especially in those using finite domains) it is possible to distinguish two execution phases from the point of view of control flow: the *programmed search* which results from the actual program steps encoded in the program clauses, and the *solver operations*, which encompass the steps performed inside the solver when adding equations or when calling the (generally built-in) enumeration predicates. These two phases can be freely interleaved during the execution. The control behavior of common CLP languages during the programmed search is similar to that of LP languages, in the sense that failure causes backtracking to the nearest branch not yet taken, and many ideas regarding visualization of control can be interchanged among them; as we will see later, in some cases the solver operations can also be shown as a specialized, implicit search procedure[8].

In general, however, understanding a visual representation of how the constraint solver works internally requires ample knowledge of its structure by the user inspecting it. A quite general tradeoff (and the one proposed here) is to leave this particular point open, and have other modules or external applications to display the data, plus the possible constraints, by using the additional, non control-related information included in the events. A starting point, both in the Herbrand and in the more general contraint solving setting, could be to visualize variables using source language constructs, although a true graphical display of variables needs a more involved representation (Section 3.11).

[8]This does not mean that such solver operations should be represented similarly to the programmed search; only that it is possible to do it that way.

3.5.1 The Programmed Search

The programmed search part of CLP execution is similar in many ways to that of LP execution. The visualization of this part of (C)LP program execution can take the form of a direct representation of the search tree, whose nodes stand for the events which take place during execution. Classical LP visualization tools, of which the Transparent Prolog Machine (TPM [EB88]) is paradigmatic, are based on this representation. In particular, the augmented And-Or tree (AORTA), used by *APT* and TPM , in which And and Or nodes are compressed and take up less vertical space, conveys basically the same information as a usual And-Or tree. Thus, an AORTA tree can be used for displaying the search as explicitly coded by the programmer, and for representing the search implicitly performed by enumeration predicates.

It is true that in CLP programs the control part has typically less importance than in LP, since most of the time is spent in equation solving and enumeration. However, note that one of the main differences between C(L)P and, e.g., Operations Research, is the ability to set up equations in an algorithmic fashion, and to search for the right set of equations. Although this part of the execution may sometimes be short and deterministic, it may also be quite large. It is, in any case, relevant for performance debugging, to be able to represent and understand the control flow.

Given the previous considerations, a first approach which can be used in order to visualize CLP executions is to represent the part corresponding to the execution of the program clauses (the programmed search) using a search tree depiction. Note that the constraint-related operations of a CLP execution (enumeration/propagation) typically occur in "bursts" which can be associated to points of the search tree. Thus, the search tree depiction can be seen as primary view or a skeleton onto which other views of the state of the constraint store during enumeration and propagation (and which we will address in Section 3.5.2) can be grafted or to which they can be related.

3.5.2 Representing the Enumeration Process as a Search Tree

The enumeration process, typically performed by finite domain solvers (involving, e.g., domain splitting, choosing paths for constraint propagation, and heuristics for enumeration), often affects performance critically. Observing the behavior of this process in a given problem (or class of problems) can help to understand the source of performance flaws and reveal that a different set of constraints or a different enumeration strategy would improve the efficiency of the program.

The enumeration phase can be seen as a search for a mapping of values to variables which satisfy all the constraints set up so far. It takes the form of (or can be modeled as) a search which non-deterministically narrows the domains of some variables and, as a result of the propagation of these changes, updates the domains of other variables. Each of these steps results in either failure (in which case another branch of the search is chosen by setting the domain of the selected variable differently, by picking another variable to update, or by backtracking to an alternative in the programmed search tree) or in a new state with updated domains for the variables.[9] Thus, one approach in order to depict this process is to use the same representation proposed for the programmed search, i.e., to use a tree representation, in either time or event space. Of course, a representation which shows all the details involved in the selection of variable and/or domain would probably convey too much info to be useful. Thus, this search can be explicitly encapsulated as nodes in the tree; these nodes should be clearly distinguished, as they represent choices of a kind different from the programmed search ones. Another alternative, which focuses more on data evolution than on control flow, is to simply visualize those steps as a series of states for all the variables (as shown in Section 3.11.1 and Figures 3.31, 3.35, and 3.47), or show an altogether *ad-hoc* representation of enumeration [SA00].

The display of the enumeration process can have different degrees of faithfulness to what is mathematically more accurate[10] and to what actually happens internally in the solver; moreover, as we have argued, a total representation is not always desirable. Actually, showing the internal behavior of the solver is not always possible, since in some CLP systems the enumeration and propagation parts of the execution are performed at a level not accessible from user code. This complicates the program visualization, since in order to gather data, either the system itself has to be instrumented to produce the data (as in the CHIP Tree Visualizer [SA00]), or sufficient knowledge about the solver operation must be available so that its operation can be mimicked externally in a meta-interpreter inside the visualizer, and inserted transparently between the user-perceived execution steps.

Other types of visualization concerned with the internal work performed by the solver need low-level support from the constraint solver. They are very useful for system implementors who have access to the system internals, and for the programmer who wants to really fine-tune a program to achieve superior performance in a given platform, but its own nature prevents them from being portable across platforms. Therefore we chose

[9] This enumeration can often be encoded as a Prolog-like search procedure which selects a variable, inspects its domain, and narrows it, with failure as a possible result. The inspection and setting of the domains of the variables are typically primitive operations of the system.

[10] Solvers often keep and upper approximation of the domain of the variables, especial in Finite Domains and in Intervals of Reals.

to move towards generalization at the expense of some losses, and base a more general, user-definable depiction, on simpler, portable primitives, while possible. These may not have access to all the internal characteristics of a CLP system, but in turn can be used in a wider variety of environments.

3.6 Coupling Control Visualization with Assertions

One of the techniques used frequently for program verification and correctness debugging is to use assertions which (partially) describe the specification and check the program against these assertions (see, e.g., [AM94, DNTM89, BDD$^+$97], and [DHM00] and its references). The program can sometimes be checked statically for compliance with the assertions, and when this cannot be ensured, run-time tests can automatically be incorporated into the program. Typically, a warning is issued if any of these run-time tests fail, flagging an error in the program, since it has reached a state not allowed by the specification. It appears useful to couple this kind of run-time testing with control visualization. Nodes which correspond to run-time tests can be, for example, color coded to reflect whether the associated check succeeded or failed; the latter case may not necessarily mean that the branch being executed has to fail as well. This allows the programmer to easily pinpoint the state of the execution that results in the violation of an assertion (and, thus, of the specification) and, by clicking on the nodes associated to the run-time checks, to explore for the reason of the error by tracking the source of instantiation of the variables. As mentioned before, the design of the tree is independent from the constraint domain, and so the user should be able to click on a node and bring up a window (perhaps under the control of a different application) which shows the variables / constraints active at the moment in which the node was clicked. This window allows the programmer to peruse the state of the variables and detect which are the sources of the bindings of the variables involved in the faulty assertion.

This does not mean, of course, that assertion checking at compile time should be seen as opposed to visualization: rather, visualization can be effectively used as user interface, with interesting characteristics of its own, to assertion-based debugging methods.

3.7 Abstracting Control

The AORTA search tree gives a good representation of the space being traversed. It also offers some degree of abstraction with respect to a classical search tree by reusing some of the tree nodes during backtracking. But it has the drawback, shared with the classical search trees, of being too explicit, taking up too much space, and showing too much detail to be usable in medium-sized computations, which can easily generate thousands of nodes. A means of abstracting this view is desirable.

An obvious way to cope with a very large number of objects (nodes and links) in the limited space provided by a screen is using a virtual canvas larger than the physical screen (as done by *APT*). However, this makes it difficult to perceive the "big picture". An alternative is simply squeezing the picture to fit into the available space; this can be made uniformly, or with a selection which changes the compression ratio in different parts of the image (this, in fact, is related to whether a time- or event-oriented view is used). The former has the drawback that we lose the capability to see the details of the execution when necessary. The latter seems more promising, since there might be parts of the tree which the user is not really interested in watching in detail (for example, because they belong to parts of the program which have already been tested).

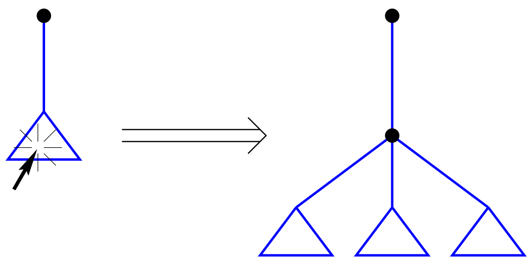


Figure 3.14: Exposing hidden parts of a tree

An example of a tool which compresses automatically parts of the search tree is the VISTA tool for the visualization of concurrent logic programs [Tic92]. This compression is performed automatically at the points of greater density of objects—near the leaves. But this disallows blowing up those parts if a greater detail is needed. An alternative possibility is to allow the user to slide virtual magnifying lenses, which provide with a sort of fish-eye transformation and give both a global view (because the whole tree is shrunk to fit in a window) and a detailed view (because selected parts of the tree are zoomed out to greater detail). Providing at the same time a compressed view of the whole search tree, in which the area being zoomed is clearly depicted, can also help to locate the place we are looking at; this option was already present in *VisAndOr*, where the canvas could be zoomed out, and the window on it was represented as a dotted square in a reduced view of the whole execution.

Another possibility to avoid cluttering up the display is to allow the user to hide parts of the tree (see Figure 3.14 and [Sch97]). This actually allows for its selective

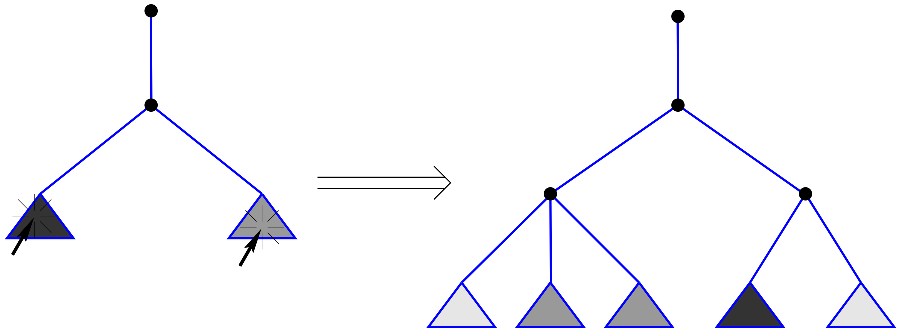


Figure 3.15: Abstracting parts of a tree

exploration (i.e., in the cases where a call is being made to a predicate known to be correct, or whose performance has already been tested). Whereas this avoids having too many objects at a time, feedback on the relative sizes of the subtrees is lost. It can be recovered, though, by tagging the collapsed subtrees with a mark which measures the relative importance of the subtrees. This "importance" can range from execution time to the number of nodes, number of calls, number of added constraints, number of fixpoint steps in the solver, etc.; different measures would lead to different abstraction points of view. Possible tagging schemes are raw numbers attached to the collapsed subtrees (indicating the concrete value measured under the subtree) or different shades of gray (which should be automatically re-scaled as subtrees are collapsed/expanded; see Figure 3.15).

3.8 Visualization for Parallel Logic Programming Systems

The abstractions basd on collapsing view are not, of course, the only possible ones in a control depiction: in any graphical representation the ultimate leading guide should be the adequacy of the pictures to what we want to explore. The *APT* control view is intimately tied to the sequential execution of logic programs; however, if we want to inspect graphically a parallel execution, probably we would be more interested in how the scheduling was performed, how long the processors were idle, etc. than in looking at the invocation of separate predicates—after all, a parallel execution is supposed not to change the semantics of the sequential program, but merely to reduce the execution time.[11] Therefore, we might devise a visualization that abstracts the source-level details

[11]There are parallel logic-based languages which do have semantics (and sometimes syntax) different from that of standard Prolog, but we will focus on parallel execution models which try to follow quite closely the sequential semantics.

of the execution while highlighting the relationships among the different branches being explored simultaneously.

Writing programs for parallel hardware has traditionally been considered a difficult task both because of the intrinsic difficulty of having to coordinate several execution threads and because of the need for considering the particular characteristics of the target machine which may also arise. On the other hand, languages which are essentially declarative, and logic languages in particular, offer great opportunities for transparently exploiting parallelism. Their well understood semantics makes them more amenable to automatic parallelization than traditional imperative languages, thus freeing the programmer from the error–prone task of data dependency analysis. One remaining problem, however, is that much of the complexity is transferred to the compiler or the program evaluation system, whose implementation then becomes quite a challenge. Furthermore, in practice, although programmers are certainly freed from worrying about low level issues, their view may be so separated from the real execution that it may be difficult for them to realize how their program is behaving. It is our belief that a clear and intuitive graphical presentation of the actual parallel execution structure at a suitable level of abstraction can greatly help both the implementor of logic programming systems and the user of such systems to achieve better results in their tasks.

Our objective now is to apply the methodology sketched in Section 3.2 and develop paradigms for the visualization of parallel models for execution of logic programs. We will focus on the visualization of Restricted And–parallelism (RAP) [DeG84, Her86a], Or–parallelism [AK90b, Lus88, CSW88, CA86] and Determinate Dependent And–parallelism (DDAP) [BHW88], although we are also interested in visualizing combinations of these models. A tool implementing such paradigms (*VisAndOr*) will be presented, and its features and usefulness illustrated through examples.

3.8.1 Common Design Concepts

Examples of common observables, aimed at studying the systems of interest in this part under the viewpoint of tasks rather than processes, are shown in Table 3.2. Given that we are focusing here on the visualization of *parallel* execution, the set of observables has been chosen to abstract away details in the sequential parts. As additional characteristics to be visualized for particular execution models are identified, new observables will be defined. For brevity the presentation will be somewhat simplified with respect to what is really present in the implementation described in Section 3.9 in terms of the number of events and the information conveyed by them.

With respect to the information contained in the events, since time is an important

Observable	Comment
START_EXECUTION	Start of the whole execution
END_EXECUTION	End of the whole execution
START_GOAL	The task (corresponding to a goal) starts
FINISH_GOAL	The task (corresponding to a goal) ends
SUSPEND	A task is suspended
RESTART	A task is restarted
FORK	Execution splits in two branches
JOIN	Different branches join

Table 3.2: Common observables for parallel execution of logic programs

issue in parallel execution, all events carry a time stamp corresponding to the time when the event occurred. This induces a natural precedence relation among events. Other types of (causal) dependencies are also present: for example, a goal can only start after its parent forks it. The conceptual graph for each execution is naturally constructed using both the time precedence and the other dependencies among events.

The visualization paradigms aim at effectively displaying the structure of the graph, as well as some information associated with each event. Temporal precedence will be assigned to spatial precedence in the vertical axis, so that the later an event is generated, the farther from the top it will be placed[12]. Despite time being the main source of precedence, it is not the only coordinate basis that we will use, as will be shown in Section 3.10. The information attached to the events will be depicted in a number of ways: for example, a different color can be assigned to each agent (or a label attached to the proper place in a monochrome display—e.g this paper).

3.8.2 Or–parallelism

Or–parallel execution corresponds to the parallel execution of different alternatives of a given predicate. It is exploited, for example, in SICStus [Swe95], Muse [Kar92, AK90a], Aurora [Lus88, Car90] and the Delphi system [CA86]. Since each alternative belongs conceptually to a different "universe", there are (in principle) no dependencies among alternatives. However, dependencies do arise in real systems due to the particular way in which common parts of alternatives are shared. Consider for example the following

[12]This orientation was chosen instead of, for example, left–to–right orientation for similarity with the usual drawings of the resolution trees.

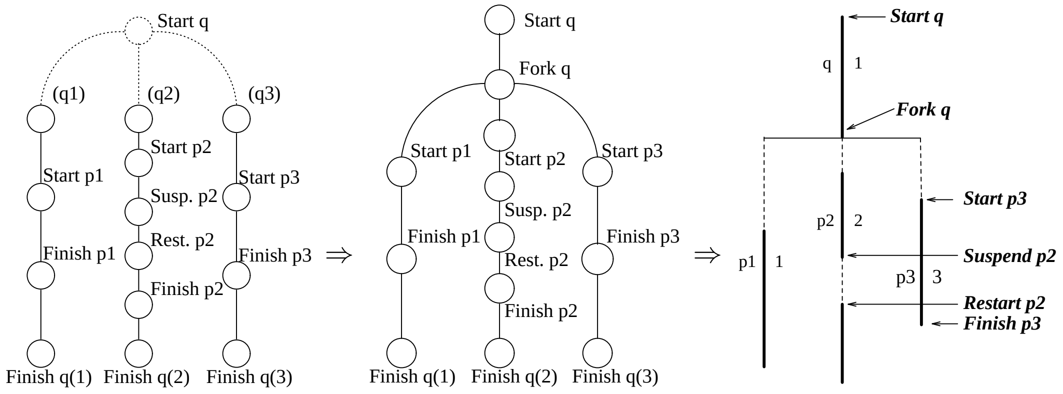


Figure 3.16: Dependency graphs for Or–parallelism and visualization

program which has three clauses for the predicate p:

```
q:- p.        p1:- ...
              p2:- ...
              p3:- ...
```

A possible dependency graph is the one depicted in Figure 3.16 left: different alternatives are represented by different *universes*. Note that p_2 is suspended at some point and then restarted. This suspension could probably have been caused by p_1, in the sense that p_2 is waiting for some built–in to be executed in p_1. In this first design we have chosen to abstract these other types of dependencies away. Many practical models share computation up to the branch point (or copy what was done before at that point). This situation is depicted in Figure 3.16, center, where a FORK has been introduced, which explicitly shows the point where execution branches. One common important feature of the dependency graphs of Or–parallel executions is that branches do not join. In terms of dependencies among observables, FORKs do not need to be balanced by JOINs. The resulting graph is thus a tree.[13]

A visualization paradigm is shown in Figure 3.16, right. The nodes of the graph have been replaced by segment starts and endings, marked with arrows in the figure.[14] Edges of the graph are represented by vertical and horizontal segments. As mentioned before actual time is represented by the vertical axis. The point where the FORK happens is marked with a horizontal thin line, whereas parallel tasks are represented as vertical

[13]Although all–solution predicates can be depicted using this paradigm, the resulting representation is not natural. A visualization closer to what the user perceives for these predicates needs structures similar to that of Restricted And–parallelism.

[14]These arrows have been added for clarification, and are not part of the visualization paradigm.

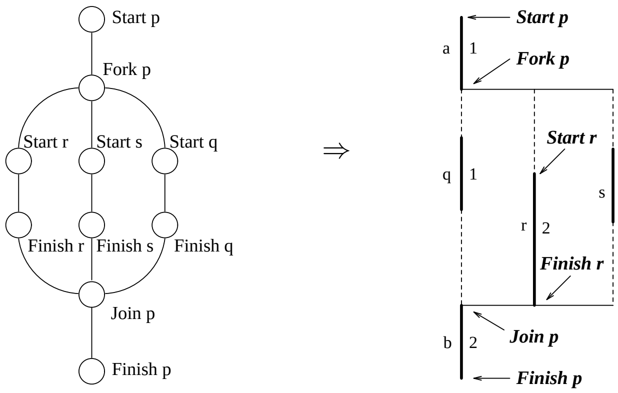


Figure 3.17: Dependency graph for Restricted And–parallelism and its visualization

lines. Each vertical thick segment represents an agent working, whereas a vertical dotted line represents a task on which no agent is working. Information associated to the events not explicitly shown in the graph is added as labels (and, if on a color display, as colors). In this case, these labels are intended to mean the clause being resolved in the branch and the agent working on it.

Real parallelism achieved can be seen simply by looking at the number of vertical thick lines present at each vertical coordinate—which represents a point in time in the execution—whereas the potential parallelism can be deduced from the total amount of vertical lines. Potential parallelism not being exploited can also be detected. Task suspension is represented by (dashed) interruptions in the vertical thick lines.

3.8.3 Restricted And–parallelism

Restricted And–parallelism (RAP), as implemented for example by &–Prolog [HG90, HG91], refers to the execution of independent goals in the body of a clause using a fork and join paradigm.[15] In this case data dependencies among the goals before and after the parallel execution and the goals executed in parallel can exist. Consider the program below, where the "&" operator, in place of the comma operator, stands for And–parallel execution:

```
p:- a, q & r & s, b.
```

[15]Non–restricted Independent And–parallelism allows execution structures which cannot be described by FORK–JOIN events. Such structures are generated, for example, by Conery's or Lin and Kumar's models [Con83, LK88] and by &–Prolog when `wait` is used. Also, similar cosntructions can be expressed using Ciao concurrency constructs.

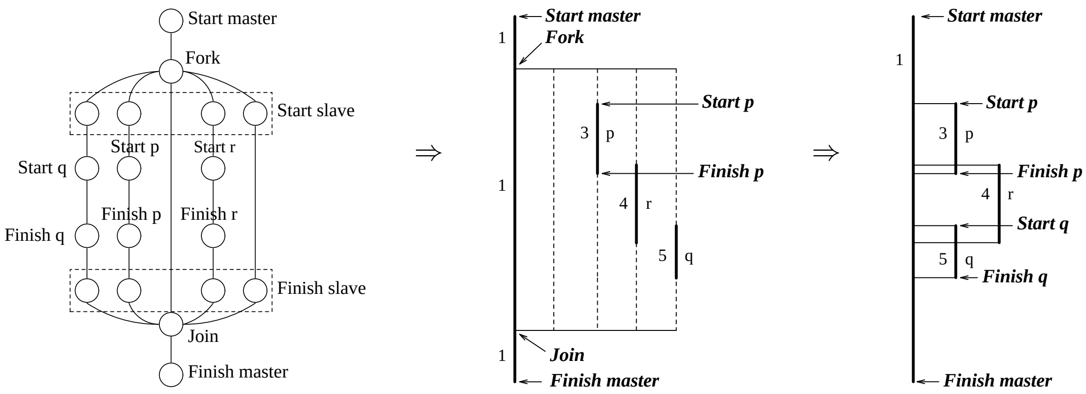


Figure 3.18: Determinate Dependent And–parallelism and two possible graphical representations

A (simplified) dependency graph for this program is depicted in Figure 3.17, left. In the RAP model there is a JOIN corresponding to each FORK (failures are not seen at this level of abstraction), and FORKS are followed by START_GOALs of the tasks originated. In turn, JOINs are preceded by FINISH_GOALs. In the case of nested FORKs, the corresponding JOINs must appear in reverse order to that of the FORKs. The START_GOAL and FINISH_GOAL events (note that finish can also be caused by ultimate goal failure) must appear balanced by pairs. Under these conditions, the RAP execution can be depicted by a directed acyclic planar graph, where And–parallel executions appear nested.

A possible visualization paradigm for RAP is shown in Figure 3.17, right. JOIN and FORK events are depicted as horizontal thin lines. The rest of the information is common with the paradigm for Or–parallelism.

3.8.4 Determinate Dependent And–parallelism

Determinate Dependent And–parallelism (DDAP) performs parallel execution when goals are detected to be determinate at run–time. There is a special agent, called the *master*, which is in charge of performing non–deterministic work. When deterministic work becomes available a number of other agents, the *slaves*, perform it in parallel. Two new observables are defined in order to notify the start and finish of a slave: START_SLAVE and FINISH_SLAVE.

The dependencies are different from those found in RAP: only one global fork is done, splitting from the master, regardless of whether there is And–parallel work available or not. In the determinate phase of the execution the master works with the slaves performing determinate reductions, and in the nondeterminate phase the master (alone) does nondeterminate reductions. START_GOAL/FINISH_GOAL events, issued by the slaves, reflect the state of each slave. A dependency graph corresponding to a possible execution is given in Figure 3.18, left.

Quite a number of visualization paradigms can be chosen for DDAP. A possible one is that illustrated in Figure 3.18, middle, in which slaves fork from the master and wait (dashed lines) for determinate work to be available. As soon as a slave starts working on a task, it becomes a solid thick line. When the task is finished, the slave becomes idle again. This has the advantage of showing every agent even if no work is being performed by it. This process–oriented representation is very similar to the traditional agent occupation charts. An alternative, task–oriented representation, which appears to be more useful in practice, is to depict the master as a thick vertical line and make slaves appear to split from it when determinate reductions are performed. Apart from producing a less crowded picture an additional reason for this choice will become clear in Section 3.8.5 since it is related to the visualization of the combination of DDAP with Or–parallelism.

3.8.5 Combinations of the Previous Types of Parallelism

Combinations of the previous schemata are possible. We will mention two of them. Or–parallelism can be combined with DDAP as in Andorra–I [SCWY91a], where Or–parallelism is not allowed under DDAP. Several master–slave teams are formed which independently work on different branches. The resulting graph is merely a tree of DDAP graphs, each of its branches being a separate universe. A similar scheme to that of Section 3.8.2 can be used for the Or–parallel part. Since various masters exist, the JOIN and FORK events must include information about their identity.

If visualization were performed by combining the fork approach for Or–parallelism (Figure 3.16, middle) and the global fork approach (Figure 3.18, middle) a tree of processor occupation charts would be obtained. But this visualization scheme is weak if slaves are allowed to migrate from a team to another team, because all slaves would, in principle, belong to every team. A good compromise would be to show only the slaves which are effectively working in a team—and this is what Figure 3.18, right, shows. Thus, we propose a combination of Figure 3.18, right and Figure 3.16, right.

Combining RAP with Or–parallelism is somewhat more tricky. AND_FORKS need to be distinguished from OR_FORKS. Allowing And–parallelism under Or–parallelism is not (conceptually) a problem, since each Or branch represents a separate universe in which And–parallelism evolves independently. The opposite situation is more complicated: allowing Or–parallelism inside And–parallelism means that multiple Or branches belonging to different And–parallel goals have to be joined. This leads, in general, to a lattice structure which is not easy to visualize in an intuitive manner. However this lattice structure can be transformed to a tree by taking a "recomputation" view of execution, as presented in [GH92].

3.9 From the Paradigm to the Tool

In this section we present *VisAndOr*, a tool which implements the visualization paradigms of the preceding sections, adding some extensions. *VisAndOr* shows statically the whole parallel execution of a logic program in a single window. The *VisAndOr* general layout is the same for all the visualization paradigms, and it also incorporates a good number of additional features beyond the simple paradigms proposed which are of much help in practice. *VisAndOr* reads event files which have previously been dumped by the system under study.[16] Figure 3.22, left, shows a window dump of *VisAndOr* and can be used for reference throughout this section.[17]

The topmost area of *VisAndOr* holds the menus, the messages, and the dialog boxes. A small window at the right always shows the whole execution. The bottommost area displays the type of parallelism being depicted and the name of the current trace file. The central part of *VisAndOr* shows the (selected part of the) execution. Time or events can be chosen as vertical measurement units. When a trace is loaded *VisAndOr* scales the execution to exactly fit the central part of the screen. In the case of complex executions, condensation is thus automatically performed by the screen resolution limitations — in fact, this is what happens, for example, in the small window at the top right corner. The scaling mentioned above can be disabled so that the time scale active before loading the trace remains active: this can be used to compare different executions.

Time or events can be measured accurately with the help of the mouse, simply by clicking and dragging to select a rectangle. The instant corresponding to the uppermost and bottommost edges of the rectangle, as well as the difference between them (measured in actual time or number of events), is shown over the menus. This rectangle, which also appears in the small window, can optionally be zoomed out to perform detailed analysis of the execution. When such a zoom is actually performed, slide bars appear surrounding the central window. Then, navigation through the picture can be

[16]In general the events are generated at a low level, so that the programs to be traced do not need to be rewritten in any way. However, the design of *VisAndOr* poses no restrictions on how the traces are to be obtained—i.e. they could also be generated using for example a meta–interpreter or a simulator.

[17]The horizontal thick line in the middle can be ignored since its meaning is totally local to the topic addressed in that picture.

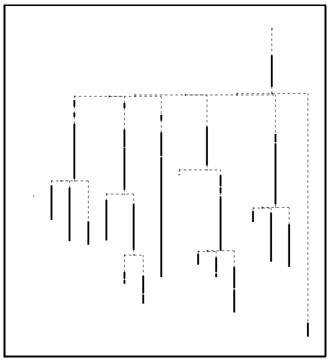


Figure 3.19: A simple Muse trace

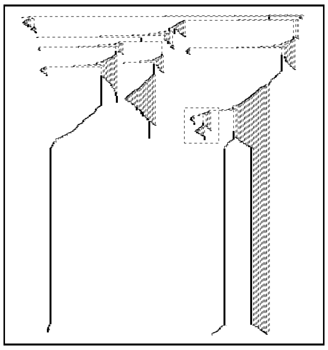


Figure 3.20: Aurora trace

accomplished using the slide bars or, alternatively, dragging the rectangle in the small window. The central window immediately responds, showing the corresponding part of the execution at the current zoom level.

VisAndOr can place icons at interesting points in the execution to give additional information about events: for example, success or failure of a branch in an Or–parallel execution. In a color display each agent is depicted in a different color. This helps to appreciate how scheduling has been performed: when scheduling favors locality in the search tree, the trace tends to have unevenly distributed colors; this is usually a better scheduling policy. Colors uniformly spread all around the execution mean the opposite situation. As suggested before (Section 3.8.1), to perform a more detailed analysis, stack set and agent identifiers can be attached to each sequential task, so that it is possible to follow their history throughout the execution[18]. As an additional help to perform scheduling analysis, a single agent's life–line can be highlighted.

VisAndOr is interfaced with the Or–parallel systems Aurora and Muse, with the Independent And–parallel system &-Prolog, and with the Determinate Dependent And+Or–parallel system Andorra–I, and is included as a third-party product in SICStus Prolog, which includes Or-parallelism capabilities imported from Muse. All these systems can generate traces which *VisAndOr* is able to understand.

3.9.1 Showing Or-parallelism

Figure 3.19 shows a simple Or-parallel Muse trace. Time is being used as measurement unit. Branch suspensions and resumptions are shown as dotted vertical lines interrupting thick vertical lines, as stated in the paradigm design. There are also delays in the starting

[18]Allowing the separate study of stack sets versus agents is mandatory in execution models where they are not intimately related.

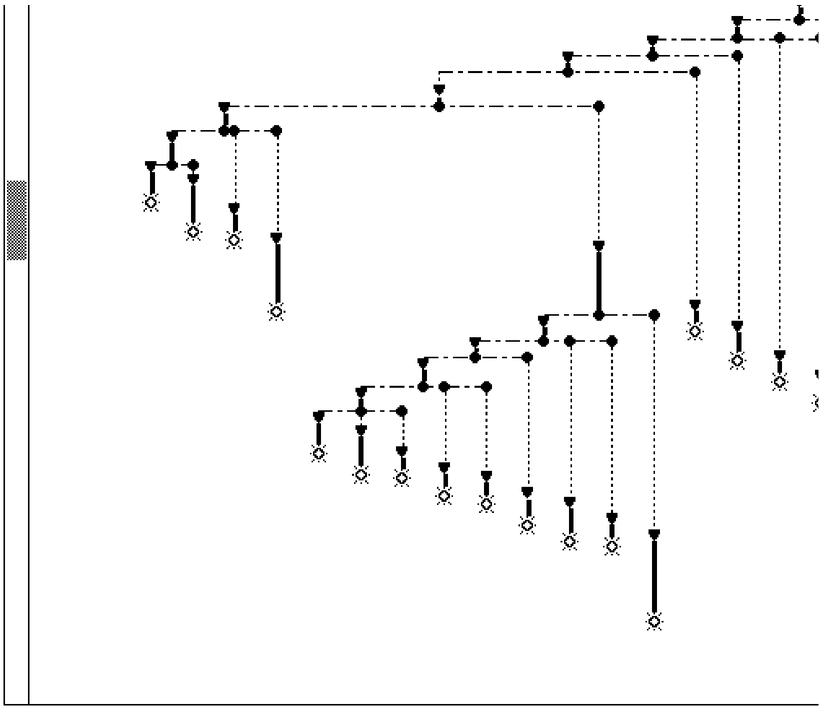


Figure 3.21: Zoom of Aurora trace

of some branches, which can be attributed (since no more information is available) to scheduling duties or perhaps to the need of waiting for built–ins in branches at the left. Another possibility is that no resources (processors) are available at this moment. This is not the case, quite obviously, but in more complicated executions realizing this is difficult. Assigning a different color to each processor greatly helps in detecting such phenomena.

Figure 3.20 shows the visual representation of a somewhat intricate Aurora execution. Even without the actual program code, it is straightforward to realize that there are three Or–parallel branches which dominate the execution. In this figure, a dashed rectangle selects a part of the execution. This part is blown up in Figure 3.21. Slide bars, which can be used to navigate through the execution, appear surrounding the execution. Icons mark points where goals are made public for parallel execution, start and finish with success.[19]

VisAndOr has been successfully interfaced with another tool based on the notion of event: the Muse Trace Tool (MUST) [SS90]. MUST has been constructed along the lines of the original WAM–Trace tool [DL87] developed at Argonne National Labs in the

[19]In fact, only success icons appear in Muse and Aurora traces, since the events are issued by the scheduler, which is, due to its design, unaware of whether a given goal failed or succeeded.

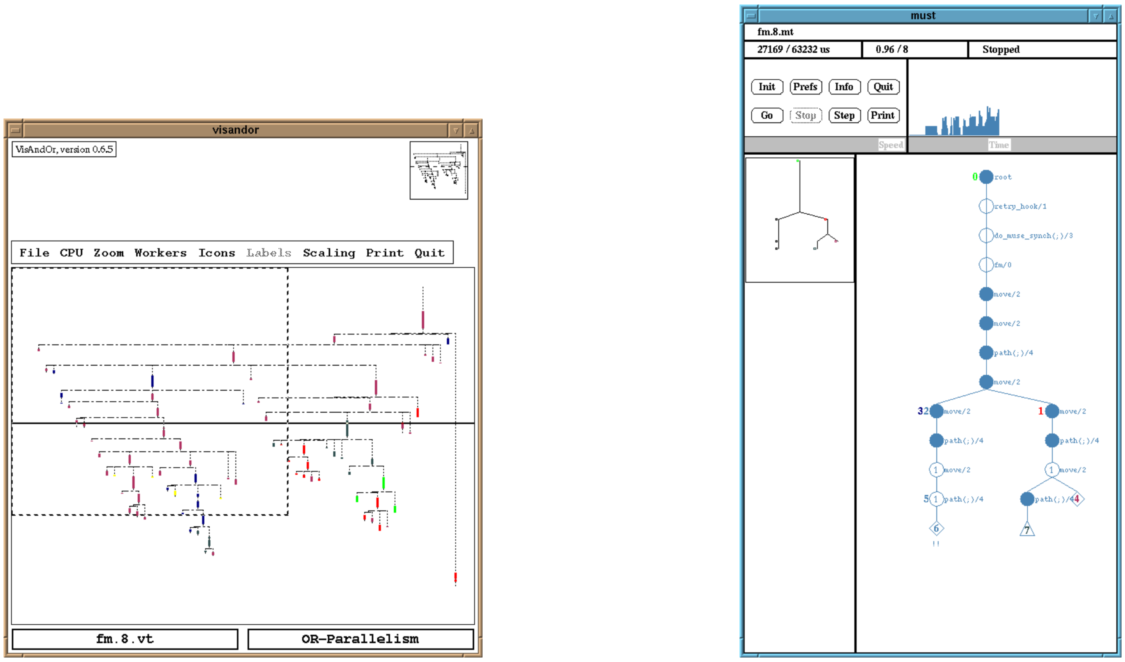


Figure 3.22: ViMust: Must and *VisAndOr* working together

context of Aurora. MUST shows snapshots of Muse executions as well as animations of such executions. The tree shown by MUST corresponds to the actual path being explored in parallel (thus it shows subgraphs of the whole graph represented by *VisAndOr*), and contains information about the state of each worker. *VisAndOr* and MUST can work together through a "standard input–output" based protocol which allows each one to send messages to the other asynchronously. *VisAndOr* indicates the point displayed by MUST with a horizontal line and answers to the messages sent by MUST to move the line. Conversely, the bar can be moved from *VisAndOr* with the mouse, and MUST receives the appropriate message to show a snapshot of the execution as required. Figure 3.22 is a snapshot of the resulting tool which has shown to be of great use at SICS. The resulting system has been given the name of ViMust. This is an example of how the use of events as an interface effectively helps integration of tools and the study of remote systems: Must traces were completely different from *VisAndOr* traces, and were translated to the desired format by a simple program.

3.9.2 Showing And-parallelism

In Figure 3.23 simple traces of a predicate with a three–branched And FORK are shown. The leftmost picture represents the predicate executed in one agent, but scheduled for

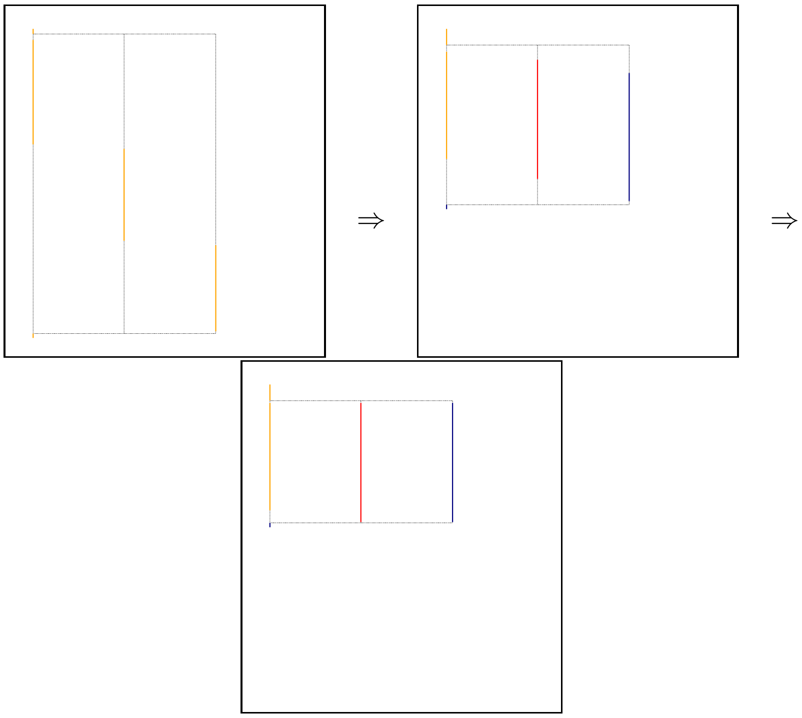


Figure 3.23: Restricted And–parallelism: sequential and parallel execution

parallel execution. Only one task is active at a time: there is only one solid line at each vertical coordinate. The figure in the middle corresponds to the same program, but executed on three agents; the time scale of the leftmost picture has been retained, so that the benefits of parallel execution in terms of time can be easily seen. Each task has a different scheduling time, as shown by the different length of the dotted vertical segments right below the FORK segment. The rightmost figure represents an ideal execution of this program, where scheduling delays have been dropped away to zero[20] [FCH96].

Figure 3.24 shows a 4x4 matrix multiplication, in one agent (leftmost picture) and four agents (middle and rightmost pictures). The recursive clauses with And–parallelism of the actual &–Prolog code look like this:

```
matrixmatrix([Vector1|Matrix1], Matrix2, [Vector3|Matrix3]):-
        matrixvector(Matrix2, Vector1, Vector3) &
        matrixmatrix(Matrix1, Matrix2, Matrix3).

matrixvector([Vector1|Matrix1], Vector2, [Scalar|Vector3]):-
        vectorvector(Vector1, Vector2, Scalar) &
```

[20]This trace was automatically obtained using the *IDRA* tool.

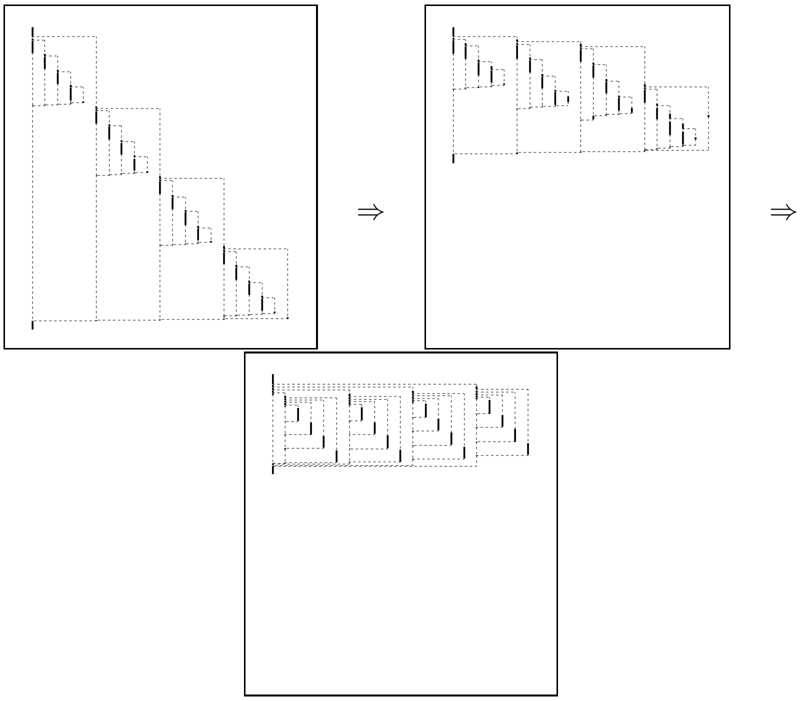


Figure 3.24: Restricted And–parallelism: nested structure and different scheduling policies

```
matrixvector(Matrix1, Vector2, Vector3).
```

The vector by vector scalar multiplication has been performed sequentially—a sort of granularity control. This example will show how scheduling can be studied with *VisAndOr*. The leftmost picture has been executed in only one agent; the structure is clearly visible. The picture in the middle shows the same program, executed in four agents. Since the `matrix_vector/3` goals are executed in a stack set different than the one they were created in (`matrix_vector/3` goals are created from left to right, as recursive goals are picked up), we can infer that recursion steps are, in this example, executed by different agents. A better scheduling, in which the agent which is executing a clause also picks up the recursive goal, is shown in the rightmost picture. This last execution runs about three times as faster as the sequential one. An utopian execution would achieve a speedup of four, but there are clearly visible sequential delays, imposed by scheduling and recursion steps, which impede this performance. A programmer debugging and tuning a scheduler would greatly appreciate this kind of feedback, which shows a high–level intuitive view of the dynamic nature of scheduling, abstracting the concrete algorithm to concentrate on its actual behavior.

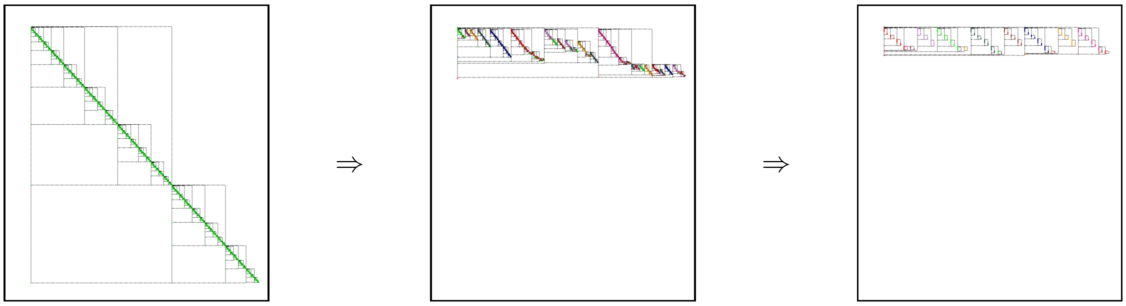


Figure 3.25: Restricted And–parallelism visualization: granularity control

Figure 3.25 shows executions of the the well–known *fibonacci* program in three different situations. From left to right, on only one agent, on eight agents without granularity control, and on eight agents with granularity control. This figure shows the tremendous impact of (a) parallel execution, and (b) granularity control. Whereas in the leftmost figure there is only one sequential execution thread, the figure in the middle shows various (up to eight) parallel tasks, but the visualization is somewhat confuse, tending to fractal[21].

This sort of executions can be cleaned up by means of granularity analysis, which tries to find out when parallel execution is *not* desirable, because scheduling costs would be bigger than the performance gained. Granularity analysis' target is to find the point where sequential execution is cheaper. By adding granularity control, so that small tasks are not scheduled for parallel execution, a remarkable speedup is obtained in Figure 3.25, right, with respect to the *naïve* parallel execution. The structure of the granularity–controlled program is much more clear than the previous one, and its execution is about twice as fast. In more complicated examples, it is difficult to perceive the impact of granularity parameters, and visualization is of much help to understand the interrelations of the different parts of the programs and their actual relative weight in the whole execution.

Figure 3.26 shows two executions of the *quicksort* program: on one processor, at left, and on four processors, at right. Apart from the speedup obtained by parallel execution, the fractal layout is evident in this example. It is interesting to compare the *fibonacci* and *quicksort* executions. Both two have repetitive patterns, but the source is somewhat different. *fibonacci* executions show mainly the structure of the algorithm, which is similar to *quicksort* in that a given problem is reduced to two simpler problems. But *quicksort* is

[21]The fractal layout is a characteristic of many RAP programs, due to the recursive character of Prolog execution.

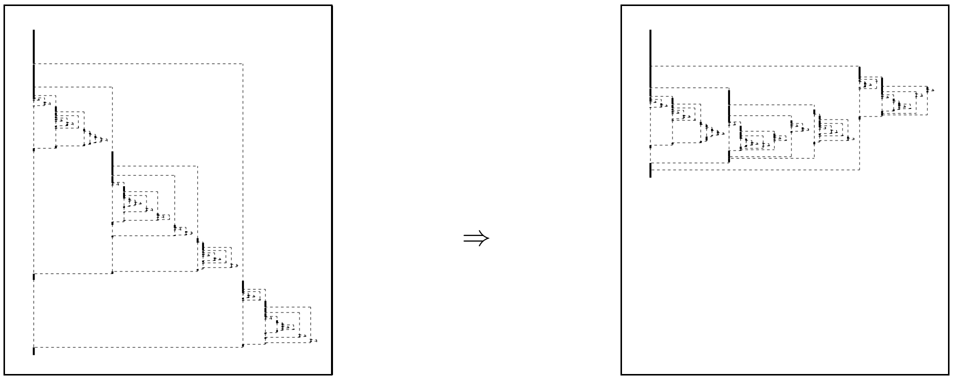


Figure 3.26: Quicksort, on 1 and 4 processors

completely *data–driven*, and its trace really reflects the initial data distribution[22].

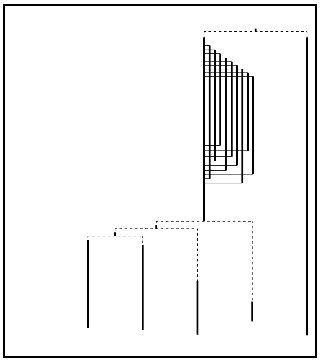


Figure 3.27: Andorra–I trace

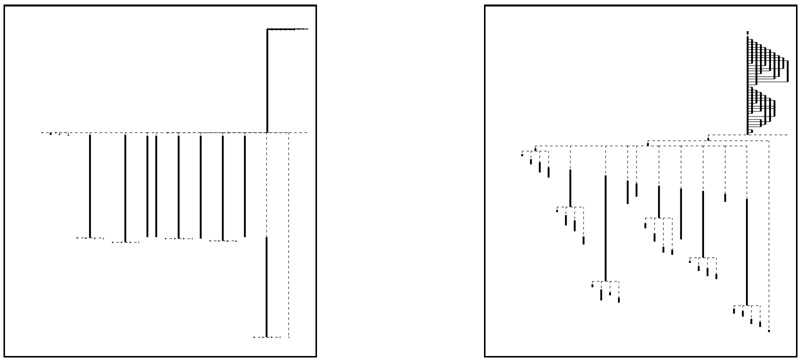


Figure 3.28: Time (left) versus events (right) in Andorra–I

[22]Of course, modulo scheduling—we can assume that an unbound number of processors are available.

3.9.3 Showing Determinate Dependent And-parallelism

Figure 3.27 shows a simple Andorra–I trace. At the top the execution splits in two Or–parallel branches; one of them has, in turn, Determinate Dependent And work and more Or–parallelism after this work is finished. The other branch corresponds to a sequential execution. Or–parallel tasks' birth is easy to appreciate, as well as waiting times before actual work. As mentioned before, two–level scheduling in Andorra–I can be visualized with *VisAndOr* by the colors mechanism and slaves visualization paradigm. The *VisAndOr* visualization paradigm can be used to understand intuitively the impact of checking determinacy conditions at run–time (as Andorra–I does): events could be associated with the start and end of this checking and signaled, for example, with icons. Its relative importance (in terms of execution time) versus the parallelism achieved can be evaluated by simply having a look at the pictures.

Figure 3.28 shows two views of the same execution. The leftmost one corresponds to the vision in the *time space*, whereas the rightmost one corresponds to the same execution in the *events space*. Valuable details about the structure of the execution which were previously hidden appear now: short executions with high parallel activity are now given more relevance than before, so allowing the perception of potential scheduling/correctness problems which otherwise would be very difficult to appreciate.

The capability of *VisAndOr* of switching to an *event space*, in which every event in the execution (say, the creation, the start, and the end of a task, among others) takes the same amount of space is aimed at showing the structure of the execution: Note that in this event-oriented view the structure of the execution is easier to see, but the notion of time is lost—or, better, traded off for an alternate view. This event-oriented visualization is the one usually portrayed in the tree-like representation for the execution of logic programs: events are associated to the CALLs made in the program, and space is evenly divided among those events. Thus, event- and time-based visualization are not exclusive, but rather complementary to each other, and it is worth having both in a visualization tool aimed at program debugging.

Additionally, as pointed out before, providing a time-aware visualization for sequential execution would provide valuable information about which parts of the program are more eager—specially when using paradigms, such as CLP, where built-in operations cannot be decomposed at user level.

3.9.4 Implementation Details

VisAndOr is written in C and runs under the X–Window environment. It has been constructed using the Xt library and Athena Widgets. They have been found to be useful, but sometimes the lack of flexibility when defining graphical objects became a problem. The inner structure of the program is quite modular: each feature is accessed through a call back routine activated by the corresponding button or menu item. The execution events are internally stored in a virtual space which is mapped to the real screen when a change of scale or representation units is requested. This can lead to problems when zooming small regions, due to the lack of virtual memory in some old X–Window servers.

One problem which was identified in previous versions of *VisAndOr* was that the algorithm to assign the space for the different branches exhausted the numerical accuracy of the computer. Of course, this happened in branches that were already indistinguishable in the screen. But errors could in some cases be carried up to higher levels, giving a wrong appearance to the whole picture. A possible solution could have been to give up computing when the branches were too near to be separated in the screen. This was not a good solution, since, on one hand, we wanted the algorithm to work in a virtual, unlimited space, unaware of the screen resolution, and, on the other hand, this would finally fail if a zoom were requested. A solution based on infinite precision arithmetic was discarded as too computationally expensive. Fortunately, a quick algorithm which did not employ at all floating point arithmetic was devised and implemented, which allow us to study traces much more complex than it had been possible previously, so assessing the effectiveness of the tool in real cases.

The difficulty of adapting the emulators and schedulers to dump traces is not, in general, very high in general, although in some systems, accurate measurements of time became a problem. In particular the *&–Prolog* implementation on Sun Sparc posed a problem which can also happen in other architectures. Time has to be consulted when an event is to be recorded. Unlike machines like the Sequent Balance, in which the time used to be stored in a memory position, so that consulting it was very quick, other OS need a system call to be performed. This system call could take a sizable proportion of the total process time, thus seriously impacting time stamps accuracy. This effect was balanced by estimating how long each system call lasted and subtracting it from the actual time. Time stamps always refer to “wall clock” time, since process time cannot be used to establish a precedence among events generated by different parallel processes.

3.10 A More General View of Events

We have reported on two tools aimed at showing the execution of sequential and parallel logic programs. We have based their design in the notion of events, either generated by a metainterpreter and saved incore (as in *APT*), or generated by the system being traced and saved to a external file. Both approaches have some assumptions about the role of events and what they are useful for, but these assumptions can be relaxed to expand the use of events.

Time-space and Event-space: In the previous sections we have assumed the vertical axis to represent time, which is useful for many purposes. However, we have also found it very convenient to enumerate sequentially the events, respecting their precedence in time, and use this number as the vertical coordinate. This gives a different, interesting view, which is very helpful in the cases in which the structure of the execution is more important than its duration, because in this view fragments of the execution which have high activity in a short time are given more relevance that long periods of sequential execution. This is the view which, by default, is provided by *APT* for sequential executions, and we can see here the usefulness of the ability to switch the view from being time-based to be event-based.

In our experience, tree-based representations such as those of the TPM, *APT*, and similar tools are certainly quite useful in education and for correctness and performance debugging. In some cases, the shape of the search tree can help in tracking down sources of unexpected low performance, showing, for example, which computation patterns have been executed more often, or which parts dominate the execution. However, the lack of a representation of time (or, in general, of resource consumption) greatly hinders the use of simple search trees in performance debugging. And-Or trees, as those used in *APT*, do not usually depict time (or in general, resource) consumption; they need to be adorned with more information.

One approach in order to remedy this is to incorporate resource-related information into the depiction itself, for example by making the distance between a node and its children reflect the elapsed time (or amount of resource consumed). Such a representation in *time space* provides insight into the cost of different parts of the execution: in a CLP language not all user-perceived steps have the same cost, and therefore they should be represented with a different associated height. For example, constraint addition, removal, unification, backtracking, etc. can have different associated penalties for different programs, and, even for the same program, the very same operation can incur in different overheads at different points in the execution. Time-oriented views have been used in several other (C)LP visualization tools.

Unifying Different Types of Events: In fact, had the sequential events defined in the first part of the chapter had included parallelism-related informacion, the events for the parallel execution could have been easily derived from them. For example, the CALLs belonging to parallel construction could have a PARALLEL(ID) information attached, where ID is a unique identifier for the corresponding invocation to the parallel construction, and the SUCCESSes corresponding to that CALLs should carry the same PARALLEL(ID) identifier. Based on this information, a parallel execution tree could have been constructed, even without the use of a parallel machine, by using an independent program. It is true, however, that this trace would lack the time information that a real machine would provide, and which is priceless to tune, e.g., scheduling algorithms, agent suspend and wakeup policies, etc. But, on the other hand, it opens new possibilities for the use of events: events can be massaged and treated as any other kind of data to extract information.

Extending Further than Visualization: Visualization is not the only topic in which the event driven scheme is useful. Dumping data tailored to different needs is a flexible way of interfacing with different tools, each of them possibly assigning a different semantics to the same set of events. The interface between the engine being traced and the tool, dictated by the format of the events, allows event files to be generated remotely or transformed to simulate special execution conditions.

As an example, the events currently recorded for *VisAndOr* can be directly used for purposes other than visualization. *IDRA* (see [FCH96] and Chapter 6) uses the same traces that *VisAndOr* does, but with a different purpose. *IDRA* finds out the optimal agent allocation and the corresponding speedup for a given parallel execution and a given number of agents. A new trace corresponding to that scheduling can be generated, which can in turn be visualized with *VisAndOr*. The speedup obtained with this trace, compared with the one obtained in a real parallel system, is a valuable indication of the quality of the actual scheduling algorithm.

Other tools adopt a similar approach: generate a trace and then examine it to extract characteristics of the execution. An example is the ParSee tool [PK96], which generates static pictures of parallel execution, portraying different metrics which give a a high-level view of the computation, such as processor occupation. These are aimed at helping the programmer to diagnose the influence on the execution of annotations for parallelism.

APT, *VisAndOr* and *IDRA* are examples of a more general approach to the design of tools aimed at monitoring and tuning other systems. Usually the target system is not to be monitored or tuned as a whole. The interesting subpart of the system under study has to be precisely defined as a *closed subsystem*, and (an abstraction of) its behavior be described in the form of observables. In order to design the observables, the following dual view can be adopted: this collection of observables plus the associated information can be taken as instructions for an abstract machine which *models* the system at the desired level of abstraction. Different models are generated by giving different semantics to these events, which give us different instances of the same concrete system.

Limits and Drawbacks of Monitoring a System: The sensitivity of the system to being traced leads us to another topic: the general approaches for monitoring programs, such as those proposed in [EJ99] and its references. That paper supports the use of a general, high level primitive, whose implementation is system-independent, and which can be used in many contexts and for a variety of goals—all of them being instances of the common target of monitoring executions. When declarative properties are the ones to be monitored, the use of this primitive poses, in principle, no problem. It has been shown [DN00b] that this approach can compete with a low-level instrumentation while giving more flexibility. However, when the main characteristic to be studied is time (or any other characteristic actually connected with the external world), the mere use of such a primitive can disturb the phenomena up to the point of rendering the observations meaningless. Thus, we still advocate the use of a low-level instrumentation when dealing with certain kind of properties. This problem will show up, although in a less acute form, when trying to investigate the performance of CLP solvers (Sections 3.12 and 3.13.3).

To Trace or Not To Trace: In practical terms, the use of traces has, in general, a drawback: the interaction with the tool is quite limited. The information gathered should include all the relevant data which could possibly be needed for a faithful replay. This aproach makes trial and error (e.g., let the user change some parameter and foresee how the execution would continue) impossible to implement in general. The only way to allow the user to interact with an execution is by generating the events (or whichever communication means is used) on-the-fly. This is the approach we have followed in the next visualization tool.

As mentioned previously, program visualization has focused classically on the representation of program flow or on the data manipulated by the program and its evolution as the program is executed.[23] We have so far explored how logic-based programs can

[23]The representation of this data usually takes advantage of invariants in the data structure (e.g., the data

be visualized, both in the control part and in the data representation part. Regarding data, we have restricted ourselves to the case of Herbrand terms, due to their ubiquity in the (cosntraint) logic programming paradigm, but we tried to make it clear that visualizations for other types of data should be possible. In particular, when tackling CLP visualization, the control part remains basically the same as in LP, but the properties of the data may very well be very different. Two key properties of the Herbrand domain which help to have a straightforward visualization are (i) that herbrand constraints can always be put in solved normal form[24], and (ii) that the terms in the Herbrand domain have no implicit meaning, so that a textual representation can, in principle, perfectly acceptable.[25]

In this last part we will focus on methods for displaying the contents of constrained variables, the constraints among such variables, the evolution of such contents and constraints, and abstractions of the proposed depictions. For simplicity, and because of their relevance in practice, in what follows we will discuss mainly the representation of finite domain (FD) constraints and variables, although we will also mention other constraint domains and consider how the visualizations designed herein can be applied to them.

3.11 Displaying Constrained Variables

In imperative and functional programming there is a clear notion of the values that variables are bound to (although it is indeed more complex in the case of higher-order functional variables). The concept of variable binding in LP is somewhat more complex, due to the variable sharing which may occur among Herbrand terms. The problem is even more complex in the case of CLP, where such sharing is generalized to the form of equations relating variables. As a result, the value of C(L)P variables often is actually a complex object representing the fact that each variable can take a (potentially infinite) set of values, and that there are constraints attached to such variables which relate them and which restrict the values they can take simultaneously.

Textual representations of the variables in the store are usually not very informative and difficult to interpret and understand.[26] A graphical depiction of the values of the

is sorted), relationships among the basic data items, or properties of abstract models the data structure tries to replicate.

[24]Assuming that circular terms, created without occurs check, are forbidden; we will come back to this point later.

[25]This last point actually opens an interesting possibility: give the user the possibility of selecting terms in the program and tailor a special visualization, i.e., give *visual semantics* to designed terms.

[26]Also note that some solvers maintain, for efficiency or accuracy reasons, only an approximation of the values the variables can take. However, in the Herbrand domain, the equations can always be represented in

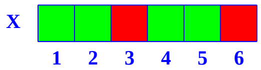


Figure 3.29: Depiction of a finite domain variable

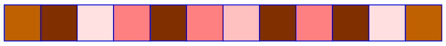


Figure 3.30: Shades representing age of discarded values

variables can offer a view of computation states that is easier to grasp. Also, if we wish to follow the history of the program (which is another way of understanding the program behavior, but focusing on the data evolution), it is desirable that the graphical representation be either animated (i.e., time in the program is depicted in the visualization also as time) or laid out spatially as a succession of pictures. The latter allows comparing different behaviors easily, trading time for space.

Since different constraint domains have different properties and characteristics, different representations for variables may be needed for them. In what follows we will sketch some ideas focusing on the representation of variables in Finite Domains, but we will also refer briefly to the depiction of other commonly used domains.

3.11.1 Depicting Finite Domain Variables

As mentioned before, Finite Domains (FD) are one of the most popular constraint domains. FD variables take values over finite sets of integers which are the domains of such variables. The operations allowed among FD variables are pointwise extensions of common integer arithmetic operations, and the allowed constraints are the pointwise variants of arithmetic constraints. At any state in the execution, each FD variable has an active domain (the set of allowed values for it) which is usually accessible by using primitives of the language. For efficiency reasons, in practical systems this domain is usually an upper approximation of the actual set of values that the variable can theoretically take. We will return to this characteristic later, and we will see how taking it into account is necessary in order to obtain correct depictions of values of variables.

A possible graphical representation for the state of FD variables is to assign a dot (or, depending on the visualization desired, a square) to every possible value the variable can take; therefore the whole domain is a line (respectively, a rectangle). Values belonging to the current domain at every moment are highlighted. An example of the representation of a variable X with current domain $\{1, 2, 4, 5\}$ from an initial domain $\{1 \ldots 6\}$ is shown in Figure 3.29. More possibilities include using different colors / shades / textures to

solved form, which makes the source language feasible to represent the store equations. This is not always possible in other domains, where a finding solved form is, in fact, one of the most difficult parts of the execution.

represent more information about the values, as in Figure 3.30 (this is done also, for example, in the GRACE visualizer [Mei96]).

Looking at the static values of variables at only one point in the execution (for example, the final state) obviously does not provide much information on how the execution has actually progressed. However, the idea is that such a representation can be associated with each of the nodes of the control tree, as suggested previously, i.e., the window that is opened upon clicking on a node in the search tree contains a graphical visualization for each of the variables that are relevant to that node. The variables involved can be represented in principle simply side to side as in Figure 3.37 (we will discuss how to represent the relations between variables, i.e., constraints, in Section 3.12).

Note that each node of the search tree often represents several internal steps in the solver (e.g., propagation is not seen by the user, or reflected in user code). The visualization associated to a node can thus represent either the final state of the solver operations that correspond to that search tree node, or the history of the involved variables through all the internal solver (or enumeration) steps corresponding to that node.

Also, in some cases, it may be useful to follow the evolution of a set of program variables throughout the program execution, independently of what node in the search tree they correspond to (this is done, for example, in some of the visualization tools for CHIP [SA00]). This also requires a depiction of the values of a set of variables over time, and the same solutions used for the previous case can be used.

Thus, it is interesting to have some way of depicting the evolution in time of the values of several variables. A number of approaches can be used to achieve this:

- An animated display which follows the update of the (selected) variables step by step as it happens; in this case, time is represented as time. This makes the immediate comparison of two different stages of the execution difficult, since it requires repeatedly going back and forth in time. However, the advantage is that the representation is compact and can be useful for understanding how the domains of the variables are narrowed. We will return to this approach later.

- Different shadings (or hues of color) can be used in the boxes corresponding to the values, representing in some way how long ago that value has been removed from the domain of the variable (see Figure 3.30, where darker squares represent values removed longer ago). Unfortunately, comparing shades accurately is not easy for the human eye, although it may give a rough and very compact indication of the changes in the history of the variable. An easier to interpret representation would probably involve adjusting the shades so that the human brain interprets

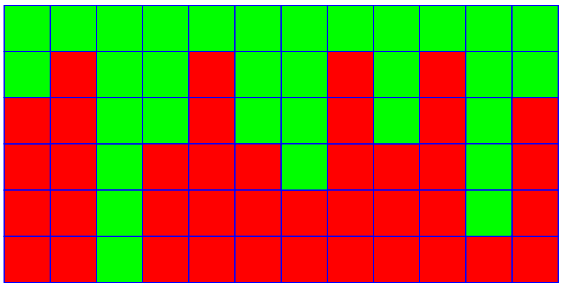


Figure 3.31: History of a single variable (same as in Figure 3.30)

```
:- use_module(library(clpfd)).
:- use_module(library(tracing_library)).

program:-
        Variables = .... ,
        Names = .... ,
        open_log(name_of_log, Variables, Names, Handle),             %% Added
        constrain_values(Variables, Handle),
        log_state(Handle),                                           %% Added
        visual_labeling(Variables, Handle),
        close_log(Handle).                                           %% Added
```

Figure 3.32: An annotated program skeleton

them correctly when squares of different shades surround it.

- A third solution is to simply stack the different state representations, as in Figure 3.31. This depiction can be easily shrunk/scrolled if needed to accommodate the whole variable history in a given space. It can represent time accurately (for example, by reflecting it in the height between changes) or ignore it, working then in *events space*, by simply stacking a new line of a constant height every time a variable domain changes, or every time an enumeration step is performed. This representation allows the user to perform an easier comparison between states and has the additional advantage of allowing more time-related information to be added to the display.

The last approach is one of the visualizations available in the *VIFID* visualizer and which, given a set of variables in a FD program, generates windows which display states

```
visual_labeling([], _Handle).
visual_labeling([Q|Qs], Handle):-
        labeling([Q]),
        log_state(Handle),
        visual_labeling(Qs, Handle).
```

Figure 3.33: The `visual_labeling/2` library predicate

(or sets of states) for those variables. *VIFID* can be used as a visualizer of the state in nodes of the search tree, or standalone, as a user library, in which case the display is triggered by spy-points introduced by the user in the program. Figure 3.32 shows an skeleton example of such an annotated program. The `open_log/4` primitive initializes the `Handle` data structure which contains the `Variables` to be observed, their `Names` and the name of the log (this can be used to save to a file, for example, or to maintain at the same time visualizations of different sets of variables and identify them using a global name). The whole domain for each variable (which is needed in later steps of the visualization) is initialized to be the domain at the time `open_log/4` is called. `close_log/1` takes the necessary actions in order to finish the visualization (e.g., closing a file, sending appropriate messages to the windows, etc.).

The actual step-by-step depiction of the current state is made by the `log_state/1` primitive. It acts as a spypoint which is placed at the points the user deems useful, and contacts with the visual side of the tool in order to communicate the current state of the variables. An important part of the CLP execution is the labeling phase, which tries to assign values to the variables compatible with the constraints put before. This labeling is usually performed by a CLP primitive, which receives a list of variables and, usually, an indication of the labeling strategy. We want, of course, to visualize the evolution of the variables during labeling. We can do this by coding this primitive so that the state of the variables is logged after each labeling step. Figure 3.33 shows an example implementation, which receives the list of variables to label and the `Handle` to the visualization and performs a tailored labeling. It is an oversimplified code for illustration purposes (we have not taken into account important issues such as the selection of variables and the labeling strategy), but it clarifies how this (and other primitives) can be interfaced with the visual tools without too much effort.

Figure 3.34 shows an annotated FD `queens` program, following the guidelines aforementioned, and Figure 3.35 shows a screen dump of a window generated by *VIFID* presenting the evolution of the (selected) program variables when solving the *queens*

```
:- use_module(library(clpfd)).
:- use_module(library(vifid)).

queens(N, Qs):-
        list_of_var_names(N, Qs, Names, 1, N),          %% Added
        open_log(queens, Qs, Names, Handle),            %% Added
        constrain_values(N, N, Qs, Handle),
        log_state(Handle),                              %% Added
        all_different(Qs),
        log_state(Handle),                              %% Added
        visual_labeling(Qs, Handle),
        close_log(Handle).                              %% Added

constrain_values(0, _N, [], _Handle).
constrain_values(N, Range, [X|Xs], Handle):-
        N > 0,
        X in 1 .. Range,
        N1 is N - 1,
        constrain_values(N1, Range, Xs, Handle),
        log_state(Handle),                              %% Added
        no_attack(Xs, X, 1).

no_attack([], _Queen, _Nb).
no_attack([Y|Ys], Queen, Nb):-
        Queen #\= Y + Nb,
        Queen #\= Y - Nb,
        Nb1 is Nb + 1,
        no_attack(Ys, Queen, Nb1).
```

Figure 3.34: The annotated `queens` FD program.

problem for a board of size 10. Each column in the display corresponds to one program variable, and it is labeled with the name of the variables on top. In this case the possible values are the row numbers in which a queen can be placed. Lighter squares represent values still in the domain, and darker squares represent discarded values. Each row in

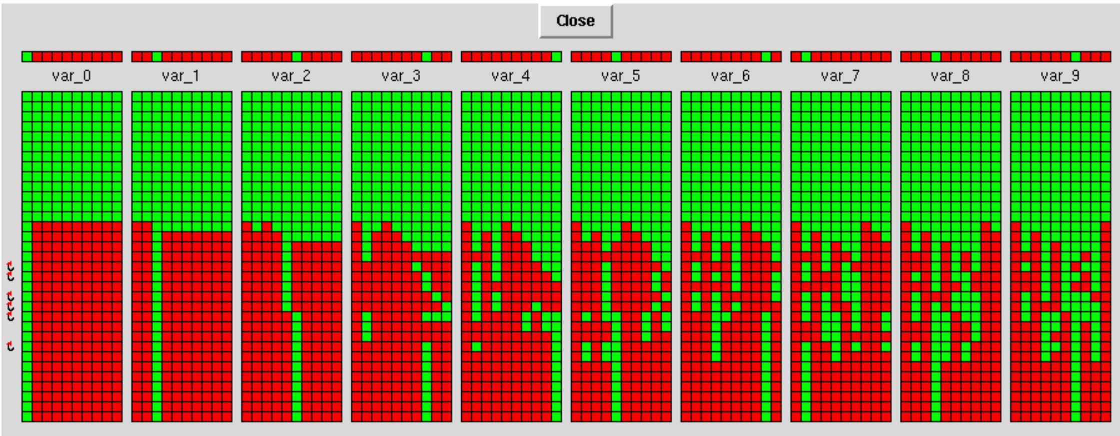


Figure 3.35: Evolution of FD variables for a 10-queens problem

the display corresponds to a spy-point in the source program, which caused *VIFID* to consult the store and update the visualization. Points where backtracking took place are marked with small curved arrows pointing upwards. It is quite easy to see that very little backtracking was necessary, and that variables are highly constrained, so that enumeration (proceeding left to right) quite quickly discarded initial values. *VIFID* supports several other visualizations, some of which will be presented later.

Some of the problems which appear in a display of this type are the possibly large number of variables to be represented and the size of the domain of each variable. Note that the first problem is under control to some extent in the approach proposed: if the visualization is simply triggered from a selected node in the search tree, the display can be forced to present only the relevant variables (e.g., the ones in the clause corresponding to that node). In the case of triggering the visualization through spy-points in the user program, the number of variables is under user control, since they are selected explicitly when starting the trace (i.e., by the `open_log` call). The size of the domains of variables is more difficult to control (we return to this issue in Section 3.13.1). However, note that, without loss of generality, programs using FD variables can be assumed to initialize the variables to an integer range which includes all the possible values allowable in the state corresponding to the beginning of the program.[27] However, being able to deduce a small initial domain for a variable allows starting from a more compact initial representation for that variable. This in turn will allow a more compact depiction of the narrowing of the range of the variable, and of how values are discarded as the execution proceeds. Other abstraction means for coping with large executions are discussed in Section 3.13.1.

[27]In the default case, variables can be assumed to be initialized to the whole domain.

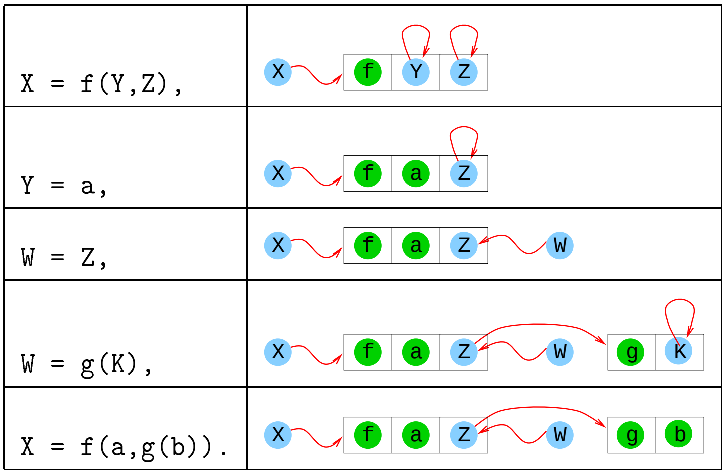


Figure 3.36: Alternative depiction of the creation of a Herbrand term

3.11.2 Depicting Herbrand Terms

Herbrand terms can always be written textually, or with a slightly enhanced textual representation. An example is the depiction of nodes in the *APT* tool. They can also be represented graphically, typically as trees. A term whose main functor is of arity n is then represented as a tree in which the root is the name of this functor, and the n subtrees are the trees corresponding to its arguments. This representation is well suited for ground terms. However, free variables, which may be shared by different terms, need to be represented in a special way. A possibility is to represent this sharing as just another edge (thus transforming the tree into an acyclic graph), and even, taking an approach closer to usual implementation designs, having a free variable to point to itself. This corresponds to a view of Herbrand terms as complex data structures with single assignment pointers. Figure 3.36 shows a representation using this view of the step by step creation of a complex Herbrand term by a succession of Herbrand constraints. Rational trees, which strictly speaking are not Herbrand terms, can also be represented in a similar way—but in this case the graph can contain cycles, although it cannot be a general graph. Systems which fully support Herbrand terms them (e.g., Prolog II, III, IV) have adopted a special textual representation for them. Other systems which feature unifications which generate rational trees do not provide a sound textual representation for them, giving up its printing when reaching a given depth in the term traversal. This can cause the user to be mislead when trying to print a term which is simply too deep.

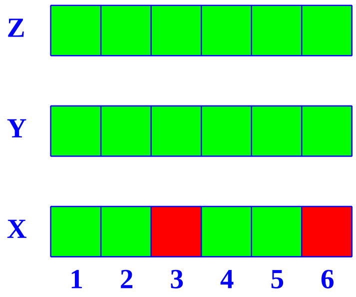


Figure 3.37: Several variables side to side

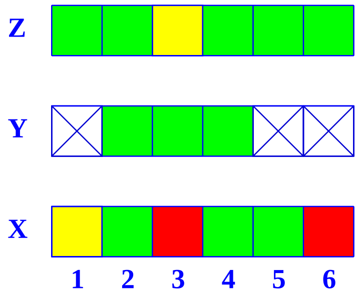


Figure 3.38: Changing a domain

3.11.3 Depicting Real Intervals

In a broad sense, intervals resemble finite domains: the constraints and operations allowed in them are analogous (pointwise extensions of arithmetic operations), but the (theoretical) set of values allowed is continuous, which means that an infinite set of values are possible, even within a finite range. Despite these differences, visual representations similar to those proposed for finite domains can be easily used for interval variables, using a continuous line instead of a discrete set of squares. An important difference between intervals and finite domains is that intervals usually allow non-linear arithmetic operations for which a solution procedure is not known, which forces the solvers to be incomplete. Thus, the visualization of the actual domain[28] will in general be an upper approximation of the actual (mathematical) domain. As a result, an exact display of the intervals is not possible in practice. But the approach for showing the evolution in time (Figure 3.35) and for representing the constraints (Section 3.12) is still valid, although in the former case some means for dealing with phenomena inherent to solving in real-valued intervals (e.g., slow convergence of algorithms) should be taken.

3.12 Representing Constraints

In the previous section we have dealt with representations of the values of individual variables. It is obviously also interesting to represent the relationships among several variables as imposed by the constraints affecting them. This can sometimes be done textually by simply dumping the constraints and the variables involved in the source code representation. Unfortunately, this is often not straightforward (or even possible in some constraint domains), can be computationally expensive, and provides too much level of detail for an intuitive understanding. Additionaly, constraints are a highly abstract

[28]Not only the representation, but also the internal storage, from which the graphical depiction is drawn.

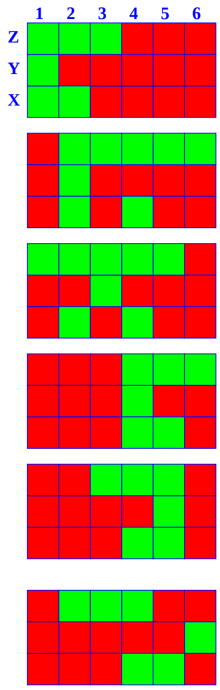


Figure 3.39: Enumerating Y, representing solver domains for X and Z

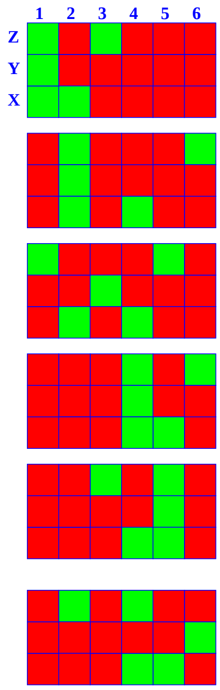


Figure 3.40: Enumerating Y, representing also the enumerated domains for X and Z

concept which only check whether it is satisfied or not, and this satisfaction is hardly understood without the constrained object or a representation thereof, i.e., a variable in a programming language. Moreover, in general there are multiple states of the variables which meet the restrictions imposed by the constraints.

Constraint visualization can be used alternatively to provide information about which variables are interrelated by constraints, and how these interrelations make those variables affect each other. Obviously, classical geometric representations are a possible solution: for example, linear constraints can be represented geometrically with dots, lines, planes, etc., and nonlinear ones by curves, surfaces, volumes, etc. Standard mathematical packages can be used for this purpose. However, these representations are not without problems: working out the representation can be computationally expensive, and, due to the large number of variables involved the representations can easily be n-dimensional, with $n \gg 3$.

A general solution which takes advantage of the representation of the actual values of a variable (and which is independent of how this representation is actually performed) is to use projections to present the data piecemeal and to allow the user to update the values of the variables that have been projected out, while observing how the variables being represented are affected by such changes. This can often provide the user with an intuition of the relationships linking the variables (and detect, for example, the presence of erroneous constraints). The update of these variables can be performed interactively by using the graphical interface (e.g., via a sliding bar), or adding manually a constraint, using the source CLP language.

We will use the constraint **C1**, below, in the examples which follow:

$$\textbf{C1} : X \in \{1..6\} \wedge X \neq 6 \wedge X \neq 3 \wedge Z \in \{1..6\} \wedge Z = 2X - Y \wedge Y \in \{1..6\} \tag{3.1}$$

Figure 3.37 shows the actual domains of FD variables X, Y, and Z subject to the constraint **C1**. As before, lighter boxes represent points inside the domain of the variable, and darker boxes stand for values not compatible with the constraint(s). This representation allows the programmer to explore how changes in the domain of one variable affect the others: an update of the domain of a variable should indicate changes in the domains of other variables related to it. For example, we may discard the values 1, 5, and 6 from the domain of Y, which boils down to representing the constraint **C2**:

$$\textbf{C2} : \textbf{C1} \wedge Y \neq 1 \wedge Y < 5 \tag{3.2}$$

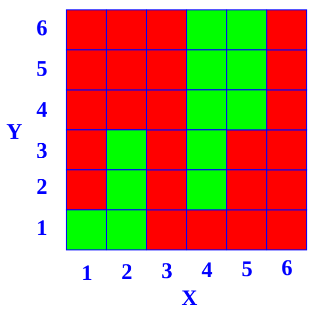


Figure 3.41: X against Y

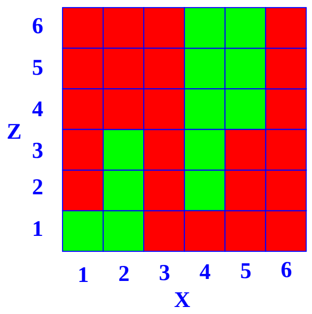


Figure 3.42: X against Z

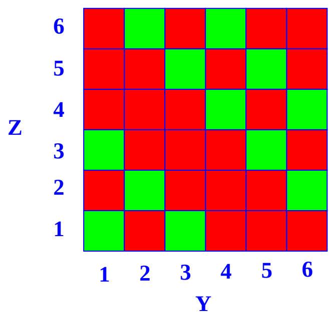


Figure 3.43: Y against Z

Figure 3.38 represents the new domains of the variables. Values directly disallowed by the user are shown as crossed boxes; values discarded by the effect of this constraint are shown in a lighter shade. In this example the domains of both X and Z are affected by this change, and so they depend on Y. This type of visualization (with the two enumeration variants which we will comment on in the following paragraphs) is also available in the *VIFID* tool.

Within this same visualization, a more detailed inspection can be done by leaving just one element in the domain of Y, and watching how the domains of X and Z are updated.

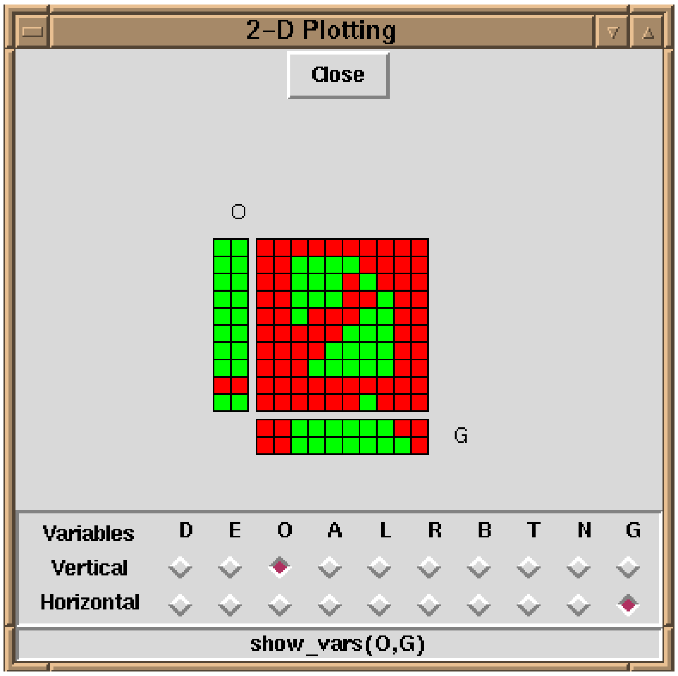


Figure 3.44: Relating variables in *VIFID*

In Figures 3.39 and 3.40 `Y` is given a definite value from `1` (in the topmost rectangle) to `6` (in the bottommost one). This allows the programmer to check that simple constraints hold among variables, or that more complex properties (e.g., that a variable is made definite by the definiteness of another one) are met.

The difference between the two figures lies in how values are determined to belong to the domain of the variable. In Figure 3.39, the values for `X` and `Z` are those kept internally by the solver, and are thus probably a safe approximation. In Figure 3.40, the corresponding values were obtained by enumerating `X` and `Z`, and the domains are smaller. Both figures were obtained using the same constraint solver, and comparing them gives an idea of how accurately the solver keeps the values of the variables. For several reasons (limitation of internal representation, speed of addition/removal of constraints, etc), quite often solvers do not keep the domains of the variables as accurately as it is possible. Overconstraining a problem may then help in causing an earlier failure. On the other hand, overconstraining increments the number of constraints to be processed and the time associated to this processing. Comparing the solver-based against the enumeration-based representation of variables helps in deciding whether there is room for improvement by adding redundant constraints.

A static version of this view can be obtained by plotting values of pairs of variables in a 2-D grid, which is equivalent to choosing values for one of them and looking at the allowed values for the other. This is schematically shown in Figures 3.41, 3.42, and 3.43, where the variables are subject to the constraint **C2**. In each of these three figures we have represented a different pair of variables. From these representations we can deduce that the values `X = 3` and `X = 6` are not feasible, regardless the values of `Y` and `Z`. It turns out also that the plots of `X` against `Y` and `X` against `Z` (Figures 3.41 and 3.42) are identical. From this, one might guess that perhaps `Y` and `Z` have necessarily the same value, i.e., that the constraint `Z = Y` is enforced by the store. This possibility is discarded by Figure 3.43, in which we see that there are values of `Z` and `Y` which are not the same, and which in fact correspond to different values of `X`. Furthermore, the slope of the highlighted squares on the grid suggests that there is an inverse relationship between `Z` and `Y`: incrementing one of them would presumably decrement the other—and this is actually the case, from constraint **C1**. A *VIFID* window showing a 2-D plot appears in Figure 3.44; the check buttons at the bottom allow the user to select the variables to depict.

Note that, in principle, more than two variables could be depicted at the same time: for example, for three variables a 3-D depiction of a "Lego object" made out of cubes could be used. Navigating through such a representation (for example, by means of rotations and *virtual tours*), does not pose big implementation problems on the graphical side, but it may not necessarily give information as intuitively as the 2-D representation. The usefulness of such a 3-D (or n-D) representation is still a topic of further research—but 3-D portraits of other representations are possible; see Section 3.13.2. On the other hand, we have found very useful the possibility of changing the value of one (or several) variables not plotted in the 2-D grid, and examine how this affects the values of the current domains of the plotted variables.

3.13 Abstraction for Constraint Visualization

While representations which reflect all the data available in an execution can be acceptable (and even didactic) for "toy programs," it is often the case that they result in too much data being displayed for larger programs. Even if an easy-to-understand depiction is provided, the amount of data can overwhelm the user with an unwanted level of detail, and with the burden of having to navigate through it. This can be alleviated by *abstracting* the information presented. Different abstraction levels and/or techniques can in principle be applied to any of the aforementioned graphical depictions, depending on which property is to be highlighted.

Note that the depictions presented so far already incorporate some degree of abstraction: when using *VIFID*, the user selects the interesting variables and program points via the spy-points and the window controls. If it is interfaced with a tree representation tool (as, for example, the one presented in this chapter), the variables to visualize come naturally from those in the selected nodes. In what follows we will present several other ideas for performing abstraction, applied to the graphical representations we have discussed so far. Also, some new representations, which are not directly based on a refinement of others already presented, but which can be thought of as abstracting constrained variables, will be discussed.

3.13.1 Abstracting Values

While the problem of the presence of a large number of variables can be solved, at least in part, by the selection of interesting variables (a task that is difficult in itself), another problem remains: in the case of variables with a large number of possible values, representations such as those proposed in Section 3.11.1 can convey information too detailed to be really useful. At the limit, the screen resolution may be insufficient to assign a pixel to every single value in the domain, thus imposing an aliasing effect which would prevent reflecting faithfully the structure of the domain of the variable. This is easily solved by using standard techniques such as a canvas that is larger than the window, and scrollbars, providing means for zooming in and out, etc. A "fish-eye" technique can also be of help, giving the user the possibility of zooming precisely those parts which are more interesting, while at the same time trying to keep as much information as possible condensed in a limited space. However, these methods are more "physical" approaches than true conceptual abstractions of the information, which are richer and more flexible.

An alternative is to perform a more semantic "compaction" of parts of the domain. As an example of such a compaction, which can be performed automatically, consider associating consecutive values in the domain of a variable to an interval (the smallest one enclosing those values) and representing this interval by a reduced number of points. A coarser-level solution, complementary to the graphical representation, is to present the domain of a variable simply as a number, denoting how many values remain in its current domain, thus providing an indication of its "degree of freedom". A similar approach can be applied to interval variables, using the difference between the maximum and minimum values in their domains, or the total length of the intervals in their domains.

Another alternative for abstraction is to use an application-oriented filtering of the variable domains. For example, if some parts of the program are trusted to be correct, their effects in the constraint store can be masked out by removing the values already discarded from the representation of the variables, thus leaving less values to be depicted.

E.g., if a variable is known to take only odd values, the even values are simply not shown in the representation. This filtering can be specified using the source language—in fact, the constraint which is to be abstracted should be the filter of the domain of the displayed variables.

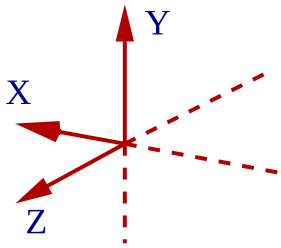
Note that this transformation of the domain cannot be completely automated easily: the debugger may not have any way of knowing which parts of the program are trusted and which are not, or which abstraction should be applied to a given problem. Thus, the user should indicate, with annotations in the program [CLI97, BDM97] or interactively, which constraints should be used to abstract the variable values. Given this information, the actual reduction of the representation can be accomplished automatically. Warnings could be issued by the debugger if the values discarded by the program do not correspond to those that the user (or the annotations in the program) want to remove: if this happens, a sort of "out of domain" condition can be raised. This condition does not mean necessarily that there is an error: the user may choose not to show uninteresting values which were not (yet) removed by the program.

3.13.2 Domain Compaction and New Dimensions

Besides the problems in applications with large domains, the static representations of the history of the execution (Figure 3.35) can also fall short in showing intuitively how variables converge towards their final values, again because of the excess of points in the domains, or because an execution shows a "chaotic" profile. The previously proposed solution of using the *domain size* as an abstraction can be applied here too. However, using raw numbers directly in order to represent this abstraction to the user is not very useful because it is not easy for humans to visualize arrays of numbers. A possible solution is to resort to shades of gray, but this may once again not work too well in practice: deducing a structure from a picture composed of different levels of brightness is not straightforward, and the situation may get even worse if colors are added.

A better option is to use the number of active values in the domain as coordinates in an additional dimension, thus leading to a 3-D visualization. A possible meaning of each of the dimensions in such a representation appears in Figure 3.45. As in Figure 3.35, two axes correspond to time and selected variables: time runs along the **Z** direction, and every row along this dimension corresponds to a snapshot of the set of FD variables which have been selected for visualization. In each of these rows, the size of the domain of the variable (according to the internal representation of the solver) is depicted as the dimension **Y**.

Figure 3.46 shows a CLP(FD) program for the `DONALD + GERALD = ROBERT` puzzle;



X The CLP(FD) variables.

Y An abstraction of the variable: the size of its domain.

Z Time.

Figure 3.45: Meaning of the dimensions in the 3-D representation.

```
:- use_module(library(clpfd)).
:- use_module(library(trifid)).

dgr(WhichOrder, ListOfVars):-
    ListOfVars = [D,O,N,A,L,G,E,R,B,T],
    order(WhichOrder, ListOfVars, OrderedVars),        %% Added
    open_log(dgr, ListOfVars, Handle),                 %% Added
    domain(OrderedVars, 0, 9),
    log_state(Handle),                                 %% Added
    D #> 0,
    log_state(Handle),                                 %% Added
    G #> 0,
    log_state(Handle),                                 %% Added
    all_different(OrderedVars),
    log_state(Handle),                                 %% Added
    100000*D + 10000*O + 1000*N + 100*A + 10*L + D +
    100000*G + 10000*E + 1000*R + 100*A + 10*L + D #=
    100000*R + 10000*O + 1000*B + 100*E + 10*R + T,
    log_state(Handle),                                 %% Added
    visual_labeling(OrderedVars, Handle),
    close_log(Handle).

order(1, [D,O,N,A,L,G,E,R,B,T], [D,G,R,O,E,N,B,A,L,T]).
order(2, [D,O,N,A,L,G,E,R,B,T], [G,O,B,N,E,A,R,L,T,D]).
```

Figure 3.46: The annotated `DONALD + GERALD = ROBERT` FD program.

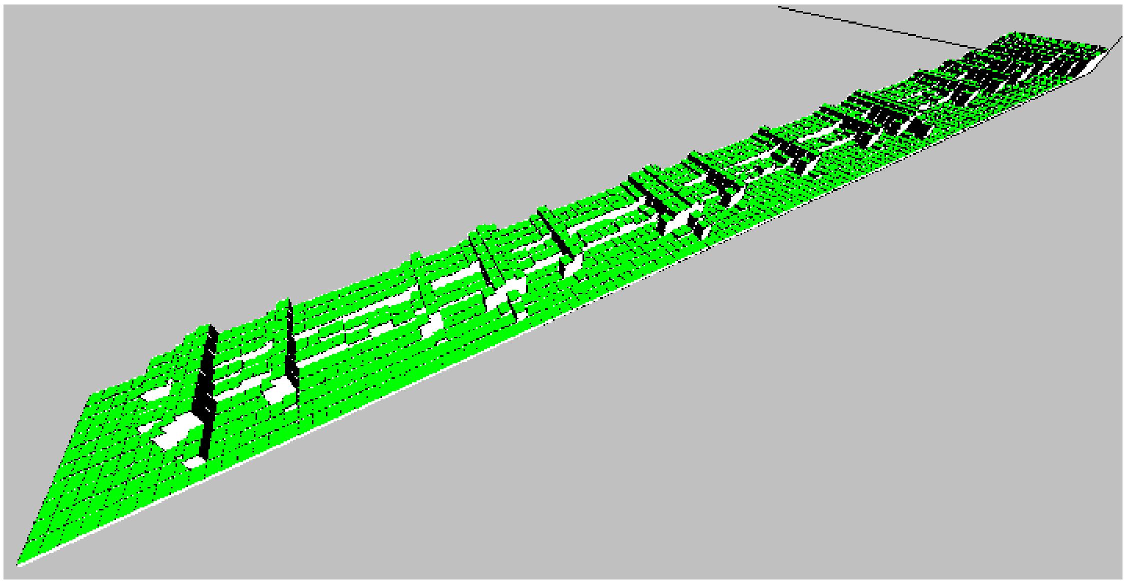


Figure 3.47: Execution of the `DONALD + GERALD = ROBERT` program, first ordering.

we will inspect the behavior of this program using two different orderings, defined by the `order/3` predicate. A different series of choices concerning variables and values (and, thus, a different search tree) will be generated by using each of these options. The program (including the labeling routines) was annotated with calls to predicates which act as spy-points, and log the sizes of the domains of each variable (and, maybe, other information pertaining to the state of the program) at the time of each call. This information is unaffected by backtracking, and thus it can also keep information about the choices made during the execution. The `Handle` used to log the data may point to an internal database, an external file, or even a socket-based connection for on-line visualization or even remote debugging.

Figure 3.47 is an execution of the program in Figure 3.46, using the first ordering of the variables. The variables closer to the origin (the ones which were labeled first) are assigned values quite soon in the execution and they remain fixed. But there are backtracking points scattered along the execution, which appear as blocks of variables protruding out of the picture. There is also a variable (which can be viewed as a white strip in the middle of the picture) which appears to be highly constrained, so that its domain is reduced right from the beginning. That variable is probably a good candidate to be labeled soon in the execution. Some other variables apparently have a high interdependence (at least, from the point of view of the solver), because in case of backtracking, the change of one of them affects the others. This suggests that the behavior

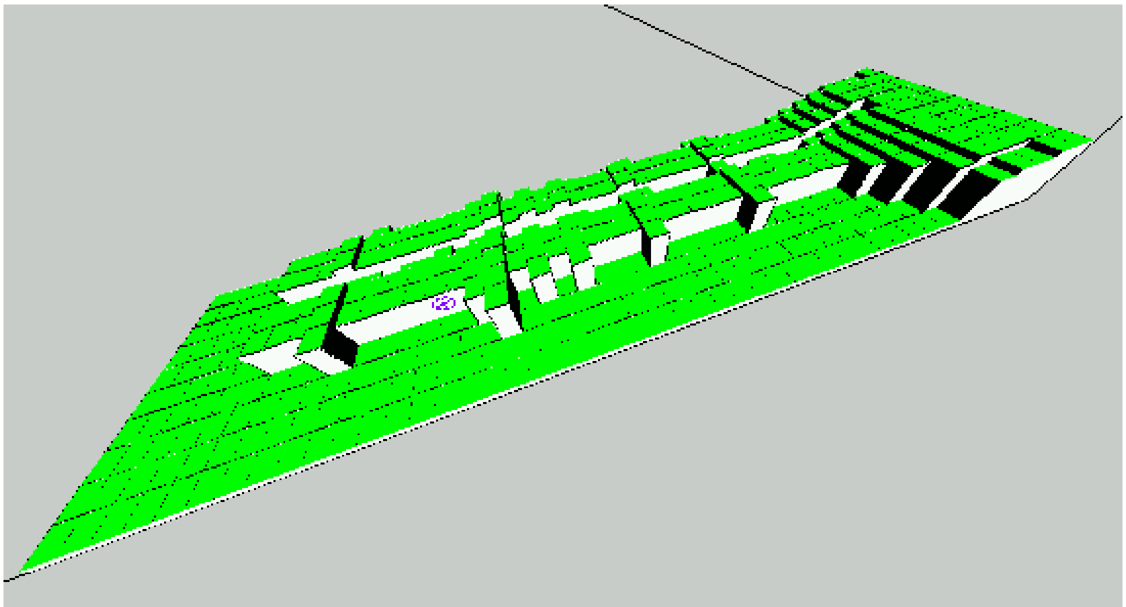


Figure 3.48: Execution of the `DONALD + GERALD = ROBERT` program, second ordering

of the variables in this program can be classified into two categories: one with highly related variables (those whose domains change at once in the case of backtracking) and a second one which contains variables relatively independent from those in the first set.

Another execution of the same program, using the second ordering, yields the profile shown in Figure 3.48. Compared to the first one, there are fewer execution steps, but, of course, the classification of the variables is the same: the whole picture has the same general layout, and backtracking takes place in blocks of variables.

These figures have been generated by a tool, *TRIFID*, integrated into the *VIFID* environment. They were produced by processing a log created at run-time to create a VRML depiction with the ProVRML package [SCH99], which allows reading and writing VRML code from Prolog, with a similar approach to the one used by PiLLoW [CH97]. One advantage of using VRML is that sophisticated VRML viewers are readily available for most platforms. The resulting VRML file can be loaded into such a viewer and rotated, zoomed in and out, etc. Additionally, the log file is amenable to be post-processed using a variety of tools to analyze and discover characteristics of the execution, in a similar way as in [FCH96]. Another reason to use VRML is the possibility of using hyper-references to add information to the depiction of the execution without cluttering the display. In the examples shown, every variable can be assigned a hyperlink pointing to a description of the variable. This description may contain pieces of information such as the source name of the variable, the actual size of its domain at that time, a profile of the changes

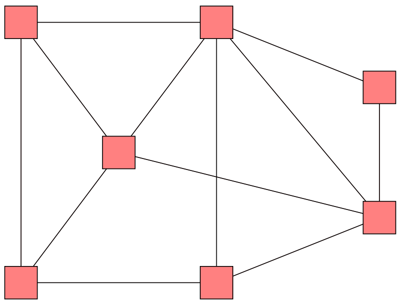


Figure 3.49: Constraints represented as a graph

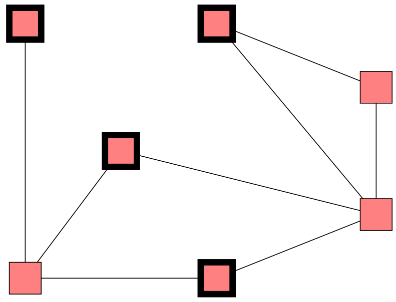


Figure 3.50: Bold frames represent definite values

undergone by that particular variable during the execution, the number of times its domain has been updated, the number of times backtracking has changed its domain, etc. Using the capability of VRML for sending and receiving messages, and for acting upon the receipt of a message, it is possible to encode in the VRML scene an abstraction of the propagation of constraints as it takes place in the constraint solver, in a similar way as the variable interactions depicted by *VIFID*.

3.13.3 Abstracting Constraints

As the number and complexity of constraints in programs grow, if we resort to visualizing them as relationships among variables (e.g., 2-D or 3-D grids plus sliding bars to assign values for other variables, as suggested in Section 3.12), we may end up with the same problems we faced when trying to represent values of variables, since we are building on top of the corresponding representations. The solutions suggested for the case of representation of values are still valid (fish-eye view, abstraction of domains, . . .), and can give an intuition of how a given variable relates to others. However, it is not always easy to deduce from them how variables are related to each other, due to the lack of accuracy (inherent to the abstraction process) in the representation of the variables themselves.

A different approach to abstracting the constraints in the store is to show them as a graph (see, e.g., [MR91] for a formal presentation of such a graph), where variables are represented as nodes, and nodes are linked iff the corresponding variables are related by a constraint (Figure 3.49)[29]. This representation provides the programmer with an approximate understanding of the constraints that are present in the solver (but not

[29]This particular figure is only appropriate for binary relationships; constraints of higher arity would need hypergraphs.

exactly *which* constraints they are), after the possible partial solving and propagations performed up to that point. Moreover, since different solvers behave in different ways, this can provide hints about better ways of setting up constraints for a given program and constraint solver.

The topology of the graph can be used to decide whether a reorganization of the program is advantageous; for example, if there are subsets of nodes in the graph with a high degree of connectivity, but those subsets are loosely connected among them, it may be worth to set up the tightly connected sections and making a (partial) enumeration early, to favor more local constraint propagation, and then link (i.e., set up constraints) the different regions, thus solving first locally as many constraints as possible. In fact, identifying sparsely connected regions can be made in an almost automatic fashion by means of clustering algorithms. For this to be useful, a means of accessing the location in the program of the variables which appears depicted in the graph is needed. This can as well help discover unwanted constraints among variables—or the lack of them.

More information can be embedded in this graph representation. For example, weights in the links can represent various metrics related to aspects of the constraint store such as the number of times there has been propagation between two variables or the number of constraints relating them. The weights themselves need not be expressed as numbers attached to the edges, but can take instead a visual form: they can be shown, for example, as different degrees of thickness or shades of color. Variables can also have a tag attached which gives visual feedback about interesting features, namely the actual range of the variable, or the number of constraints (if it is not clear from the number of edges departing from it) it is involved in, or the number of times its domain has been updated.

The picture displayed can be animated and change as the solver proceeds. This can reflect, for example, propagation taking place between variables, or how the variables lose their links (constraints) with other variables as they acquire a definite value. In Figure 3.50 some variables became definite, and as a result the constraints between them are not shown any more. The reason for doing so is that those constraints are not useful any longer: this reflects the idea of a system being progressively simplified. It may also help to visualize how backtracking is performed: when backtracking happens, either the links reappear (when a point where a variable became definite is backtracked over and a constraint is active again in the store), or they disappear (when the system backtracks past a point where a constraint was created).

Further filtering can be accomplished by selecting which types of constraints are to be represented (e.g, represent only “greater than” constraints, or certain constraints flagged in the program through annotations). This is quite similar to the domain filtering proposed in Section 3.13.1.

3.14 Implementation Details

VIFID and *TRIFID* are implemented as Prolog libraries which provide primitives for opening a log, logging a state, and closing a log. They are implemented in Prolog and Tcl/Tk, and rely on a few primitives to open socket connections and to spawn and communicate with other processes (primarily for the Tcl/Tk part).

VIFID is completely interactive, and the library is aware of the changes to the selected variables by having the program variables accessible from the `Handle` which materializes the communication on the Prolog side. Calls to `log_state` inspect the domain of these variables in order to draw the depictions. Since the library has direct access to the same variables the program uder study is manipulating, the user can update (monotonically) the domain of the variables being selected from the graphical interface.[30] The execution can continue after updating, but the user does not have to commit to this update: a RESET button forces the program to go back to the point where the update was made. To implement this, the tool pushes a *named choicepoint* (i.e., a choicepoint whose address is accesible from the Prolog implementation, a very useful characteristic found in several Prolog implementations) and later, if needed, backtracks to it, or discards it.

The visualization part of *VIFID* is completely written in Tcl/Tk, and the commands were issued by the Prolog library. Only a few rutines commonly used throughout the execution were written directly in Tcl/Tk and called from Prolog. The flexibility of Tcl/Tk was enough, since most of the windows have a simple layout. The speed of Tcl/Tk was not much of a problem, except when the number of objects in the window became very large (e.g., several thousand, which is possible when the variable history visualization is selected). The 2-D visualization was computationally expensive, as it has to enumerate the pairs of variables and check for satisfiability. Overall, the tool was strong enough to be used routinely, and the visualization was found to be useful and easy to understand.

TRIFID shares many ideas with *VIFID*: it is also a Prolog library which scans the variables it has access to. But instead of starting an interactive visualization, we decided to take advantage of a Prolog to VRML interface build for Ciao Prolog and generate VRML for the visualization. Gathering the data was not a computational problem; instead, we found troubles related with the size of the files being generated[31], and with the

[30] Actually, the user is able to call any Prolog goal from *VIFID*.

[31] More precisely, the VRML visualizers had problems with that!

speed of the VRML visualizers (freely) available at the time. Thus, we were not able to analyze executions as big as we had desired, but the pictures we obtained are intuitively understandable.

3.15 Related Work

The work in visualization in Computer Science is very ample, even if we restrict to work directly related to Logic Programming. Thus, we will mention only the work directly related to performance and correctness debugging. Visuazations tools aimed at achieving, e.g., more friendly development environments, visualization aimed at teaching, will not be mentioned, unless an important part of them tackles directly debugging matters.

3.15.1 *APT* and other Work in Visualization of Sequential Execution

The paradigmatic example of visualization of sequential execution is the Transparent Prolog Machine [EB88], developed at the Open University with educational aims. It displays the execution of a Prolog program using AORTA trees and shows the input and output modes of the arguments to the call.

Color Prolog [NKD97, KP96], also with an educational focus, displays the whole execution tree and traces the history of the variables, assigning a different color to each variable. Conflicts during unification was solved using the so-called “color unification”.

The Mozart Oz system includes a visualizer for searth trees which allows expanding and collapsing subtrees [Sch97]. The search rules are not fixed: some are provided by the system, and users can write their own search rules. As it focuses mainly on the control part, it does not provide an explicit means for displaying different types of data (e.g., constraints, constrained variables, propagation steps).

Search tree views have been studied and implemented in [DA00]. The tool is geared towards facilitating the understanding of the behavior of programs, especially in the case of complex combinatorial systems. The originality lies in its generality: the computation space can be analyzed with several views which are separately specified in the same languae as the program. The abstractions are specified using program predicate properties, as suggested in Section 3.13.1.

The Prolog IV [PRO] development system has a visual debugger [Bou00] based upon the Byrd Box model which dispatches information to several viewers, which are complementary of each other. The Execution Tree Viewer gives different capabilities within a search tree, namely a full view of the execution tree, the view of a particular proof, and the replay of the execution to a given point in the search tree.

3.15.2 *VisAndOr* and Other Related Visualization Tools for Parallel Programming

The ParaGraph tool[32] by Aikawa *et al.* [AKK+92] is aimed at tuning the Parallel Inference Machine (PIM) [GSN+88]. It is aimed to perform low–level (processor–oriented) and high–level (goal–oriented) profiling. ParaGraph gathers data during program execution using primitives of the KL1 [UC91] language. It is not a general purpose tool, but rather a highly machine and language dependent tool.

The WAMTrace tool [DL87] is a visualization tool for Or parallel full Prolog, originally written for ANLWAM, and later used for Aurora. This tool shows an animation of the parallel search tree, with different icons being extensively used to reflect different worker states and node types. The main difference with *VisAndOr* is that *VisAndOr* offers a static and much more schematic view, conveying the whole execution. A similar viewpoint is offered by MUST. The extensive use of animated icons provide a dynamic stream information; this approach could be of interest to represent suspension in DAP and in Constraint Logic Languages.

The VISTA tool [Tic92] intends to give effective visual feedback to a programmer tuning concurrent logic programs. Procedure invocations are displayed radially from the root with explicit condensation at the leaves (if this is needed). The drawings obtained with VISTA have the peculiar shape of a snail shell, due to the mapping of the (parallel) search tree into a polar coordinate system. This system, which represents Deterministic Dependent And–parallelism, is, in some ways, similar to *VisAndOr*'s forerunner, VisiPal [HN90].

VACE [VPG97] is a tool developed in the context of the ACE project [PGH95, GHPSC94], which offers a visualization of And/Or parallelism based on *recomputation trees*, designed to distinguish clearly the And- and the Or-parallel branches. Or-parallel alternatives within an And-parallel call are shown as *leaves* of a book from which other (possibly parallel) computations start. It is also based on analyzing a post-mortem trace generated by the system under study. The trace is a superset of that used by *VisAndOr*, and thus VACE is able to display *VisAndOr* traces.

The VisAll tool [FIVC98] tries to unify the different formats and views of other previous tools. A modular design allows having*translators* for different trace formats (including those of VACE and *VisAndOr*). The traces are used to construct an *execution graph*, and several output components give different portrays of the execution, including processor occupation graphs and other statistics. It is probably the most mature and

[32]There are two different visualization tools with the same name: the one we are currently referring to and the ParaGraph tool by Heath and Etheridge, described in [HE91], which are very different and must not be confused.

complete tool for parallel logic program visualization.

Finally, in the realm of the LOGFLOW project [Kac93, Kac92, Kac97, Kac94], some graphical monitoring tools have been implemented [KZP97]. These are aim mainly at tuning parameteres related to distributed scheduling, token passing, and granularity control, than to a depiction of the search tree. This is, of course, due to the very nature of the LOGFLOW system.

3.15.3 Related Work in Contraint Visualization

Early work in constraint visualization was made in the realm of the Eclipse [ECR93] system; the GRACE system[Mei96] represented the values of constrained variables as we did in Section 3.11.1. The representation was connected to a Byrd box model for program debugging, and it was used to show values of variables. Additional information, such as how long ago values were removed from the domain of a variable, was encoded using different color shades.

More recently, the DisCiPl project [DHM00] fostered the use of visualization and assertion-based debugging tools. Within this effort, several visualization tools (some of them already mentioned) were developed.

The Search Tree Visualization tool [SA00] built for the CHIP FD programming system focuses on displaying the search tree created when labeling finite domain variables, despite its name. It is heavily influenced by the capabilities of the Oz Explorer [Sch97] (e.g., ability to choose the search strategy), but tries to add facilities not present in Oz (e.g., representation of contraint propagation steps). The Search Tree visualizer is able to represent the history of the variables under a number of different views, and to replay the execution as needed. A recomputation-based technique is used to perform that: when the execution is performed information about the shape of the tree (but not about the domain of the variables) is kept. When the user selects a node, the execution is replayed up to that point by taking the right branches at the choicepoints. The trace is generated by special labeling predicates the program is instrumented with.

Some constraint applications need to set up complex relationships among the variables which express, e.g., precedence among events or constraints on the placement of a set of items[33]. In those cases a visualization which mimics the initial problem setting helps in mapping performance/debugging problems in the constraint solving to the original problem and to the code. The Global Constraint visualization tool [SABB00] does precisely this, by incorporating special visualizations tailored to some of the complex

[33]Sometimes this is not the natural way of expressing the original problem, but a transformation allow this modelling.

constraints available in the CHIP system. Although this gives an intuitive representation, it needs the user to map the problem to one of these *standard* complex constraint templates, and the implementor to provide the necessary hooks in the constraint solving. This visualization can be classified as a application-oriented abstraction.

The visualization of constraint networks, proposed here as an cosntraint abstraction amenable of being treated and studied, was implemented in the Constraint Investigator [TM99], interfaced with the Oz Explorer [Sch97]. This proposal visualizes a graph which is close to the implementation, since the nodes of the graph can be propagators, variables, and events (which trigger the action of propagators). The ability to expand and collapse the constraint net, and to filter the variables about which we wish to display information, incrfeses the tool usefulness in the case of big executions. Overall, it gives a good representation of the store, but probably needs some further structure to represent complex problems. Besides, the store represented is intimately tied to the Oz constraint solver.

Another graph-based visualization of constraints is [Tak00], which builds a 3-D graph to debug the constraints used by a tool aimed at placing geometric objects with placement constraints. Unlike in our approach (Section 3.13.3), nodes represent both constraints and objects (thus removing the need of hypergraphs). Nodes representing constraints are boxes, and nodes representing objects are spheres. The type of constraint and object decides the color of the object. The freedom degree of an object (i.e., whether its value/placement is or not completely fixed) can be determined by animating the picture: the object is *pulled* by the system by solving the constraint system after changing the value of the associated variable. The solution is compared with that obtained previously, and the difference shown graphically.

3.16 Conclusions

We have presented some design considerations regarding the depiction of search trees resulting from the sequential execution of constraint logic programs. We have argued that these depictions are applicable to the programmed search part and to the enumeration parts of such executions. We have also presented a concrete tool, *APT*, based on these ideas. Two interesting characteristics of this tool are, first, the decoupling of the representation of control from the constraint domain used in the program (and from the representation of the store), and, second, the recording of the point in which every variable is created and assigned a value. The former allows visualizers for variables and constraints in different domains (such as those presented in Section 3.11) to be plugged in and used when necessary. The latter allows tracking the source of unexpected values and failures, and so is of use for correctness debugging. *APT* has served mainly as an experimentation prototype for, on one hand, studying the viability of some of the depictions proposed, and, on the other hand, as a skeleton on which the constraint-level views can be attached.

We have also reported on several visualization paradigms to represent the execution of logic programs. These depictions are devised to summarize and give an intuitive view of different characteristics of the programs being executed. A set of tools which implement these paradigms have been developed and tested. The visualizations developed focus on the parallel execution of logic programs, the representation of the sequential, and the evolution of variables in constraint logic programs.

Parallel execution of logic programs is displayed by *VisAndOr*. The visualization paradigm of the tool has been developed following a methodology which starts by determining at what level we want to visualize the model, what are the basic elements at this level, and which dependencies hold among them. The tool has been interfaced with a few implementations of parallel logic processing paradigms, currently &-Prolog, Aurora, Muse, Andorra-I, and SICStus, and has been actively used at Bristol, SICS and Madrid. Its usefulness as debugging and tuning tool for paral1el logic systems has been reported and assessed through practical applications. The top-down approach followed in the design of the visualization paradigm makes the tool homogeneous in the representation and in the user interface, and the event-based interface has been used in other tools. The interface with the execution platform is formally defined up to the point of being comparable to an abstract machine language, so that different semantics can be used to highlight different characteristics or to transform the execution skeleton to simulate different environments. This allows to easily extend and adapt the tool to visualize different paradigms and to easily interface it with other tools designed under similar principIes. This has been done, and the resulting tools improved their usefulness (see Chapter 6).

The work in *VisAndOr* is being currently extended [Mar00] to add new additional features in the style of the ParaGraph tool [HE91] which do not require designing new paradigms (i.e., processor utilization, real parallelism achieved versus potential parallelism present...). Future work includes new conceptual representations to support several other forms of parallelism and their combinations: among them we may cite the representation of suspension needed for Dependent And-parallel and concurrent execution models.

Last, we have discussed techniques for visualizing data evolution in CLP. The graphical representations have been chosen based on the perceived needs of a programmer trying to analyze the behavior and characteristics of an execution. We have proposed solutions for the representation of the run-time values of the variables and of the run-time constraints among them. In order to be able to deal with large executions, we have also discussed some abstraction techniques, including the 3-D rendition of the evolution of the domain size of the variables. The proposed visualizations for variables and constraints have been tested using two prototype too1s: *VIFID* and *TRIFID*. These visualizations can be easily related, so that tools based on them can be used in a complementary way, or integrated in a larger environment. In particular, in the environment that we have developed, each tool can be used independently or they can all be triggered from a search tree visualization (e.g., *APT*).

VIFID (and, to a lesser extent, *TRIFID* which is less mature) has evolved into a practical tool and is quite usable by itself as a library which can be loaded into a number of CLP systems. Also, some of the views and ideas proposed have since made their way to other tools, such as those developed by Cosytec for the CHIP system [SA00].

Abstractions for all the aforementioned visualizations have been proposed. This aims at alleviating the task of understanding the interesting characteristics of the execution of programs when many details are available in the raw pictorial depiction.

Each of these tools uses a different approach to monitoring executions. *APT* is built around a metainterpreter; *VisAndOr* reads events generated by an engine modified at low-level; and *VIFID/TRIFID* use a spypoint approach. Every tool has a reason for that:

- *APT* recorded the history of the unifications; doing this is in principle possible by a careful instrumentation of the program which increments the number of arguments in each predicate in order to carry the information ([Duc92a, Duc91] are good examples of a similar approach), and saving the collected information to be used in case of branch failure. Unfortunately executing metacalls is much more difficult, since (part of) the rewriting program must be present and used at runtime to rewrite the terms to be called. Similarly, if dynamic modification of the program is used, the assert and retract calls have to be catched and the terms rewritten. The execution of metacalls within a metainterpreter is much easier, since most of the machinery needed is already present.

- *VisAndOr* needed the traces to be generated with as little disturb to the execution as possible. Thus metainterpreting was completely dicarded, and the use of spypoints would probably not provide results accurate enough. On the other hand, the modifications to the parallel engine to generate traces were minimal.

- *VIFID* and *TRIFID* did not keep a history of the variables, and so unification (i.e., constraint solving) could be left to the CLP system. This means that a metainterpreter was not needed, and, although this approach could have been followed, previous experience with *APT* taught us that it could be too slow to tackle medium-sized executions. Moreover, the information about control flow needed to implement *VIFID* could be gathered at the Prolog level, so no low-level changes had to be done. The primitives available for exploring CLP variables were enough for our aims, and so we did not have to tackle the task of chaging the core of the CLP solver (which, on the other hand, would have probably been a quite hard task).

In retrospect, we feel that the metainterpreter approach is the more error-prone and long-lasting one, and the results in terms of speed are superseded by the others. On the other hand, if the metainterpreter is self-contained and uses basic capabilities common to many CLP systems, its portability makes it a valuable tool. This portability can in any case be partially recovered by the use of a clear, simple, well-defined set of events. This allowed, for example, to make *VisAndOr* traces understandable by many other systems.

Chapter 4

Optimizing Executions with Data Parallelism

Chapter Summary

Much work has been done in the areas of and–parallelism and data parallelism in Logic Programs. Such work has proceeded to a certain extent in an independent fashion. Both types of parallelism offer advantages and disadvantages. Traditional (and–) parallel models offer generality, being able to exploit parallelism in a large class of programs (including that exploited by data parallelism techniques). Data parallelism techniques on the other hand offer increased performance for a restricted class of programs. The thesis of this paper is that these two forms of parallelism are not fundamentally different and that relating them opens the possibility of obtaining the advantages of both within the same system. Some relevant issues are discussed and solutions proposed. The discussion is illustrated through visualizations of actual parallel executions implementing the ideas proposed.

4.1 Introduction

The term *data parallelism* is generally used to refer to a parallel semantics for (definite) iteration in a programming language such that all iterations are performed simultaneously, synchronizing before any event that directly or indirectly involves communication among iterations. It is often also allowed that the results of the iterations be combined by reduction with an associative operator. In this context a *definite iteration* as an iteration where the number of repetitions is known before the iteration is initiated.

Data parallelism has been exploited in many languages, including Fortran–90 [MR90], C* [Thi90], Data Parallel C [HQ91], *LISP [Thi86], etc. Recently, much progress has been reported in the application of concepts from data–parallelism to logic programming, both from the theoretical and practical points of view, including the design of programming constructs and the development of many implementation techniques [Vor92, NT88, BM88, Bla92, Kac90, Wis86, Mil90, Bar90, BLM93a, BLM93b].

On the other hand, much progress has also been made (and continues to be made) in the exploitation of parallelism in logic programs based on control–derived notions such as and–parallelism and or–parallelism [Con83, DeG84, DeG87, HG90, KK84, LK88, War87b, Lus90, Ali88, AK90b, GJ89, GSCYH91, GHPSC94, Fag87, Kal87b, She92a, War88, SCWY91a]. It appears interesting to explore, even if only informally, the relation between these two at first sight different approaches to the exploitation of parallelism in logic programs. This informal exploration is one of the purposes of this paper (the other being to explore the intimately related issue of fast task startup).

4.1.1 Data Parallelism and And–Parallelism

It is generally accepted that data parallelism is a restricted form of and–parallelism: [1] the threads being parallelized in data–parallelism are usually the iterations of a recursion, a type of parallelism which is obviously also supported in and–parallel systems. All and–parallel systems impose certain restrictions on the goals or threads which can be executed in parallel (such as independence and/or determinacy, applied at different granularity levels [HR95, Nai88, SCWY91a, GHM93, HC93]) which are generally the *minimal* ones needed in order to ensure vital desired properties such as correctness of results or “no–slowdown”, i.e. that parallel execution be guaranteed to take no more time than sequential execution. Data–parallel programs, since they are after all and–parallel programs, have to meet the same restrictions from this point of view. This is generally referred to as the “safeness” conditions in the context of data parallelism. Such conditions are imposed among the iterations being parallelized (examples are requiring them to be deterministic, to have only one alternative, and/or to be independent).

However, one of the central ideas in data–parallelism, as presented in many proposals, is to impose *additional* restrictions to the parallelism allowed, in order to make possible further optimizations in some important cases, in return for a certain loss of parallelism due to not being able to deal with the general case. I.e., the additional restrictions imposed have the obvious drawback that they limit the amount of parallelism which can be obtained with respect to a more general purpose and–parallel implementation. On the other hand, when the restrictions are met, many optimizations can be performed with respect to an unoptimized general purpose and–parallel model, in which the implementation perhaps has to deal with backtracking, synchronization, dynamic

[1] Note, however, that data parallelism can also be exploited as or parallelism [Pre94, CDO88].

scheduling, locking, etc. A number of implementations have been built which are capable of exploiting such special cases in an efficient way (e.g. [BLM93a, BLM93b]). The particular restrictions imposed over general purpose and–parallelism vary slightly from one proposal to another. In general, only recursions of a certain type are allowed to be executed in parallel. Also, limitations are posed on the level of nesting of these recursions (e.g. sometimes no nesting is allowed). Often, a priori knowledge of the sizes of the lists (or arrays) being operated on is required (but this data is also obtained dynamically in other cases).

In a way, one would like to have the best of both worlds: an implementation capable of supporting general forms of and (and also or) parallelism, so that speedups can be exploited in as many programs as possible, and at the same time have the implementation be able to take advantage of the optimizations present in data–parallel implementations when the conditions are met.

4.1.2 Compile–time and Run–time Techniques

In order to achieve the above mentioned goal of a "best of both worlds" system, there are two classes of techniques which have to studied. The first class is related to detecting when the particular properties to be used to perform the optimizations hold. However, this problem is common to both control– and data–parallel systems. The concept of "data parallelism" does not in any way make the task of the compiler or the implementation simpler in this regard. The solution of allowing the programmer to explicitly declare such properties or use special constructs (such as "parallel map," "bounded quantifications" [ABB93], etc.) which have built–in syntactic restrictions may help, but it is also true that this solution can be applied indistinctly in both of the approaches under consideration. Thus, we will not deal herein with how the special cases are detected.

The second class of techniques are those related to the actual optimizations realized in the abstract machine to exploit the special cases. Given, as we have argued before, that data–parallelism constitutes a special case of and–parallelism, one would in principle expect the abstract machine used in data–parallelism to be a "pared down" version of the more general machines. We believe that this is in general the case, but it is also true that the data–parallel machines also bring some new and interesting techniques.

For the sake of discussion, we will concentrate on the abstract machine of Reform Prolog [BLM93a, BLM93b]. In many aspects, the Reform Prolog abstract machine can in fact be viewed as a "pared–down" version of a general–purpose and–parallel abstract machine such as the RAP–WAM/PWAM [Her86b, HG90], the DASWAM [She92a], or the Andorra-I engine [SCWY91b]. For example, there are a number of agents or workers which are each essentially a WAM. Also, the dynamic scheduling techniques are very similar to the goal stealing method used in the RAP–WAM.

Understandably, there are also some major differences. A first class of such differences is related to the optimizations in memory management which are possible with respect to general purpose abstract machines due to the special case of and–parallelism being dealt with. For example, because of the restrictions posed on backtracking among parallel goals, structures like the "markers" of the RAP–WAM, which delimit stack sections corresponding to different goals and to different backtracking points, are not necessary (Chapter 5). However, it should be noted that the same optimizations can also be done in general–purpose abstract machines supporting and–parallelism, such as the RAP–WAM, if the particular case is identified, and without losing the general case [Her86a, TPGC94, PGH95, SH94]. Both dynamic and static detection of such special cases has been studied. A similar argument can be made regarding some other minor optimizations that, for lack of space, will not be addressed explicitly.

On the other hand, a number of optimizations, generally related to the "Reform Compilation" done in Reform Prolog [Mil91], are more fundamental. We find these optimizations particularly interesting because they bring attention upon a very important issue regarding the performance of and–parallel systems: that of the speed in the creation and joining of tasks. We will essentially devote the rest of the paper to this issue, because of the special interest of this subject, and given that, as pointed out before, the other intervening issues have already been addressed to some extent in the literature.[2]

4.2 The Task Startup and Synchronization Time Problems

The problem in hand can be illustrated with the following simple program:

```
vproc([],[]).
vproc([H|T],[HR|TR]) :-
        process_element(H,HR),
        vproc(T,TR).
```

which relates all the elements of two lists. Throughout the discussion we will assume that the `vproc/2` predicate is going to be used in the "forwards" way, i.e. a ground list of

[2]Improving the performance of and–parallel systems in the presence of fine-grained computations can also be addressed by performing "granularity control", where goals that could have been run in parallel but are too small grain are sequentialized. This is usually done by determining (statically or dynamically) the cost of goals and sequentializing them or grouping them when such cost falls below a given threshold. This very interesting issue can be treated orthogonally to the techniques that we discuss in this paper. Relevant work can be found in [DLH90, KS90, LGHD94, ZTD$^+$92] and their references.

values and a free variable will be supplied as arguments (in that order), expecting as a result a ground list.

4.2.1 The Naive Approach

This program can be naively parallelized as follows using "control–parallelism" (we will use throughout &–Prolog [HG91] syntax, where the "&" operator represents a potentially parallel conjunction):

```
vproc([],[]).
vproc([H|T],[HR|TR]) :-
        process_element(H,HR) & vproc(T,TR).
```

This will allow the parallel execution of all iterations. Note that the parallelization is safe, since all iterations are *independent*. The program can be parallelized using "data–parallelism" in a similar way.

However, it is interesting to study the differences in how the tasks are started in both approaches, due to the textual ordering of the goals. In a system like &–Prolog, using one of the the standard schedulers (we will assume this scheduler throughout the examples), the initial agent, running the call to `vproc/2`, would create a process corresponding to the recursion, i.e. `vproc(T,TR)`, make it available on its goal stack, and then take on the execution of `process_element(H,HR)`. Another agent might pick the created process, creating in turn another process for the recursion and taking on a new iteration of `process_element(H,HR)`, and so on. In the end, parallel processes are created for each iteration. Note that all process creation has been a simple consequence of the application of the parallel conjunction operator semantics. This is very attractive in that the same operator which allows parallelism among two goals in any general case, also yields in this particular case the desired result of parallelizing all the iterations of a "loop". However, the approach or, at least, the naive program presented above, also has some drawbacks.

In order to illustrate this, we perform the experiment of running the previous program in the following context. We assume a query "`?- makevector(10,V), main(V,VR).`", where `makevector(N,L)` simply instantiates `L` to a list of integers from 1 to `N`. Thus, we have a list of 10 elements. We use as `process_element/2` a small–grained numerical operation, which serves to illustrate the issue:

```
process_element(H,HR) :-
        HR is ((((H * 2) / 5)^2)+(((H * 6) / 2)^3))/2.
```

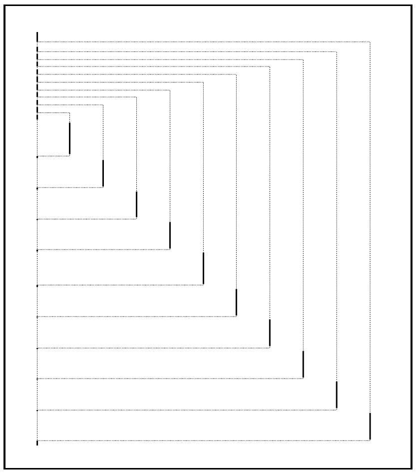


Figure 4.1: Vector operation (10 el./1 proc.)

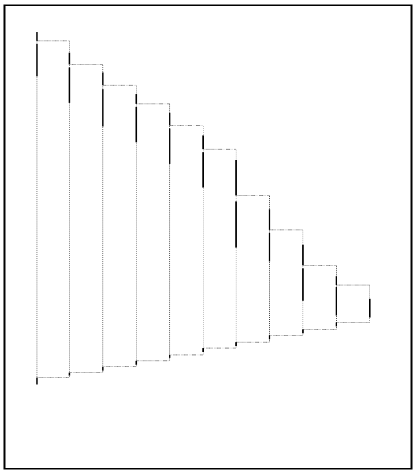


Figure 4.2: Vector operation, giving away recursion (10 el./8 proc.)

Finally, in order to observe the phenomenon, we run the program in &–Prolog on 8 processors on a Sequent Symmetry and generate a trace file, using the following predicate:

```
main(V,VR) :-
        start_event_trace,
        vproc(V,VR),
        stop_event_trace,
        save_trace('Eventfile').
```

The trace is then visualized with VisAndOr [CGH93]. In VisAndOr graphs, time goes from top to bottom. Vertical solid lines denote actual execution, whereas vertical dashed lines represent waits due to scheduling or dependencies, and horizontal dashed lines represent forks and joins. Figure 4.1 represents the execution of the benchmark in one processor, and serves as scale reference.

The result of running the benchmark in 8 processors is depicted in Figure 4.2. As can be seen, the initial task forks into two. One is performed locally whereas the other one, corresponding to the recursion, is taken by another agent and split again into two. In the end, the process is inverted to perform the joins. A certain amount of speedup is obtained; this can be observed by comparing to Figure 4.1 — the total amount of time is less. However, the speedup obtained is in fact quite small for a program such as this with obvious parallelism. This low speedup is in part due to the small granularity of the parallel tasks, and also to the slow generation of the tasks which results from giving out the recursion [CGH93].

4.2.2 Keeping the Recursion Local

One simple transformation can greatly alleviate the problem mentioned above — reversing the order of the goals in the parallel conjunction, so that the recursive goal is kept local, and not even pushed on to the goal stack:

```
vproc([],[]).
vproc([H|T],[HR|TR]) :-
        vproc(T,TR) & process_element(H,HR).
```

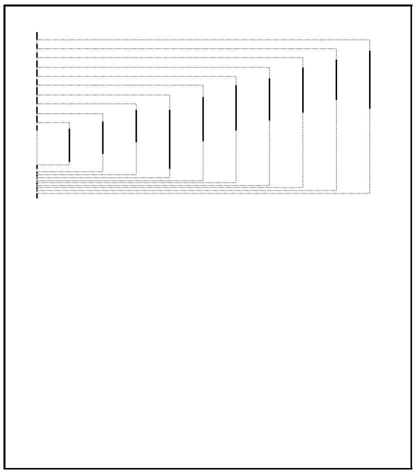


Figure 4.3: Vector operation, keeping recursion (10 el./8 proc.)

The result of running this program is depicted in Figure 4.3, which uses the same scale as Figures 4.1 and 4.2. The first process can now be observed to keep the recursion local and thus create the tasks much faster, resulting in substantially more speedup. It should be noted that this transformation is in fact in most cases done automatically by the &–Prolog parallelizing compiler. However, the compiler leaves hand–parallelized code as is and this has allowed us before to write and run the program that hands out the goals in the "wrong" way.

Keeping recursions local can speed up the process of task creation, and in most applications, which in general show much larger granularity than this example, task creation speed is not a problem. On the other hand, in numerical applications such as those targeted in data–parallelism, task creation using linear recursion will still be a problem: the speed of the process creating the tasks will become a bottleneck.

4.2.3 The “Data–Parallel” Approach

At this point it is interesting to return to the data–parallel approach and, in particular, to Reform Prolog. The way this system tackles the problem (we assume that it has already been identified that the recursion is suitable for this technique) is by first converting the list into a vector (and noting the length on the way) and then creating in a tight, low level loop the corresponding tasks, which are simply represented by a pointer to the element of the vector which the task should operate on. The following program allows us to both illustrate this process without resorting to low level instructions and measure inside &–Prolog the benefit that this type of task creation can bring (once the parallel conjunction is set up, each task creation in and–prolog in fact corresponds to pushing two pointers on to a goal stack — the overhead in the previous cases was coming from the recursion and the setup time for each parallel conjunction):

```
vproc([H1,H2,H3,H4,H5,H6,H7,H8,H9,H10],
     [HR1,HR2,HR3,HR4,HR5,HR6,HR7,HR8,HR9,HR10]) :-
        process_element(H1,HR1) &
        process_element(H2,HR2) &
        process_element(H3,HR3) &
        process_element(H4,HR4) &
        process_element(H5,HR5) &
        process_element(H6,HR6) &
        process_element(H7,HR7) &
        process_element(H8,HR8) &
        process_element(H9,HR9) &
        process_element(H10,HR10).
```

Figure 4.4 represents the same execution as Figure 4.3, but at a slightly enlarged scale; this scale will be retained throughout the rest of the paper, to allow easy comparisons of the pictures.

The result of the execution of this “data–parallel” program is depicted in Figure 4.5, which uses the same scale as Figure 4.4. The improvement is clear and due to the much faster task creation and joining (and also to having only one synchronization structure for all tasks). Note, however, that the creation of the first task is slightly delayed due to the need for unifying the whole list before creating any tasks and for setting up the tasks themselves. This small delay is compensated by the faster task creation, but can eventually be a bottleneck for very large vectors. Eventually, in a big computation with a large enough number of processors, the head unification will tend to dominate the whole

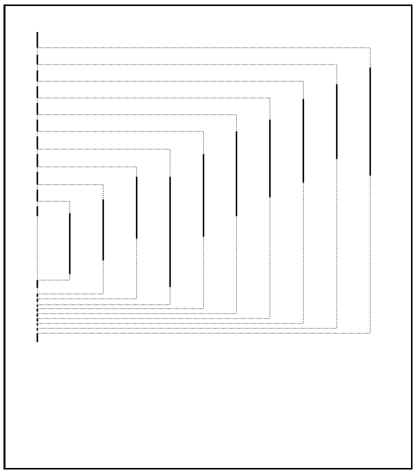


Figure 4.4: Vector operation, keeping recursion (10 el./8 proc.)

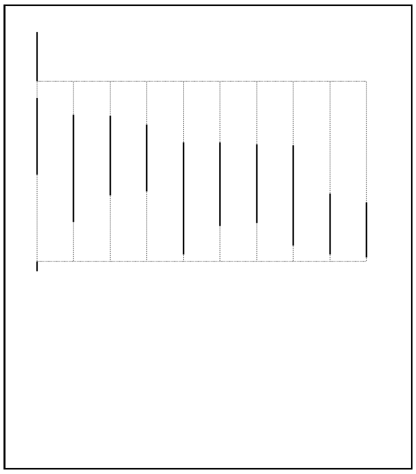


Figure 4.5: Vector operation, flattened for 10 elements (10 el./8 proc.)

computation (c.f. Amdahl's law). In this case, unification parallelism can be worthwhile [Bar90].

In our quest for merging the techniques of the data–parallel and and–parallel approaches, one obvious solution would be to incorporate the techniques of the Reform Prolog engine into the PWAM abstract machine for the cases when it is applicable. In fact, we believe that very little modification to the PWAM would is necessary, as shown in [HC96]. On the other hand, it is also interesting to study how far one can go with no modifications (or minimal modifications) to the machinery.

The last program studied is in fact a straightforward unfolding of the original recursion. Note that such unfoldings can always be performed at compile–time, provided that the depth of the recursion is known. In fact, knowing recursion bounds may actually be frequent in traditional data–parallel applications, (and is often the case when parallelizing bounded quantifications [ABB93]). On the other hand it is not really the case in general and thus some other solution must be explored.

4.2.4 A More Dynamic Unfolding

If the depth of the recursion is not known at compile time the previous scheme cannot be used. But instead of resorting directly to the naive approach, we can try to perform a more flexible task startup. The following program is an attempt at making the unfolding more dynamic, while still staying within the source–to–source program transformation approach:

```
vproc([H1,H2,H3,H4|T],[HR1,HR2,HR3,HR4|TR]) :-
        !,
        vproc(T,TR)  &
        process_element(H1,HR1) &
        process_element(H2,HR2) &
        process_element(H3,HR3) &
        process_element(H4,HR4).
vproc([H1,H2,H3|T],[HR1,HR2,HR3|TR]) :-
        !,
        vproc(T,TR)  &
        process_element(H1,HR1) &
        process_element(H2,HR2) &
        process_element(H3,HR3).
vproc([H1,H2|T],[HR1,HR2|TR]) :-
        !,
        vproc(T,TR)  &
        process_element(H1,HR1) &
        process_element(H2,HR2).
vproc([H|T],[HR|TR]) :-
        !,
        vproc(T,TR) &
        process_element(H,HR).
vproc([],[]).
```

The results are shown in Figure 4.6, which has the same scale as Figures 4.4 and 4.5. A group of four tasks is created; one of these tasks creates, in turn, another group of four. The two remaining tasks are created inside the latter group. The speed is not quite as good as when the 10 tasks are created at the same time, but the results are close.

This "flattening" approach has been studied formally by Millroth [3] [Mil90], which has given sufficient conditions for performing these transformations for particular cases such as linear recursion.

There are still two problems with this approach, however. The first one is how to chose the "reformant level", i.e. the maximum degree of unfolding used, which with this

[3]And has been used in &–Prolog compilation informally (see e.g. [WH87] and some of the standard &–Prolog benchmarks).

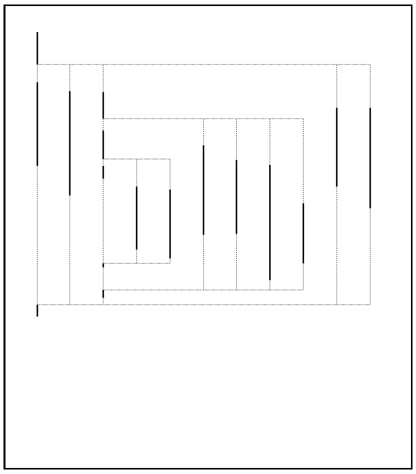


Figure 4.6: Vector operation with fixed list flattening (10 el./8 proc.)

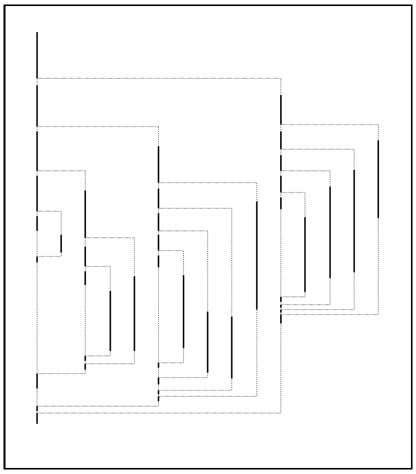


Figure 4.7: Vector operation with flexible list flattening (10 el./8 proc.)

technique is fixed at compile–time. In the previous example the unfolding was stopped at level 4, but could have gone on to a higher level. The ideal unfolding level depends both on the number of processors and the size of lists. For large lists a large unfolding may be desirable. However, the program size also grows, as well as the chain of intermediate unifications made by the last iterations. The other problem, which was pointed out before, is the fact that the initial matching of the list (or the conversion to a vector) is a sequential step which can become a bottleneck for large data sets. A solution is to increase the speed of creation of tasks, but that has a limit. In fact, it will also eventually become a bottleneck, even if low level instructions are used. Another solution is to use from the start, and instead of lists, more parallel data structures, such as vectors (we will return to this in Section 4.3).

4.2.5 Dynamic Unfolding In Parallel

We now propose a different solution which tries to address at the same time the two problems above. We give the solution for lists. The transformation has two objectives: speeding up the creation of tasks by performing it in parallel, and allowing a form of "flexible flattening". The basic idea is depicted in Figure 4.8. Instead of simply performing a unification of a fixed length as encoded at compile–time, a builtin, `skip/4`, is used which will allow performing unifications of different lengths.

The predicate `skip(L,N,LS,NS)` relates a list `L` and an "unfolding increment" `N` with a

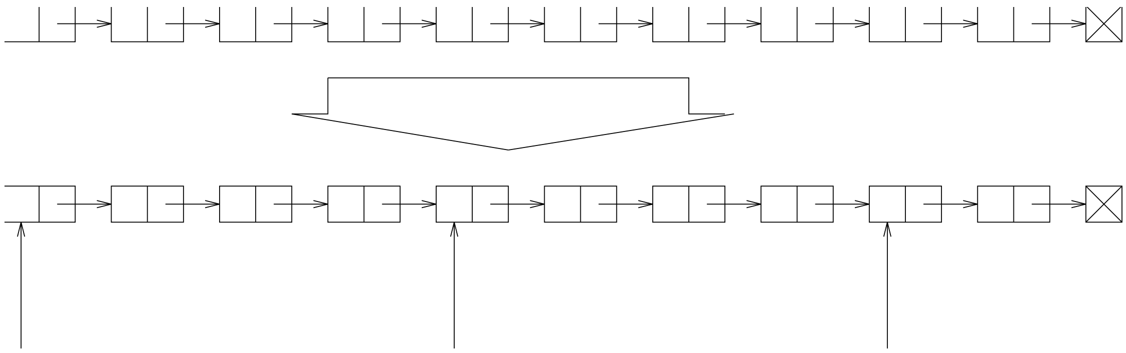


Figure 4.8: “Skip” operation, 10 elements in 4

sublist LS of L which is placed at most at N positions from the starting of L. NS contains the actual number of elements in LS, in case that N is less than the length of L (in which case LS = []). The utility of skip(L,N,LS,NS) is that several calls to it using the output list LS as input list L in each call will return pointers to equally–spaced sublists of L, until no sufficient elements remain. Figure 4.8 depicts the pointers returned by skip(L,N,LS,NS) to a 10 elements list, with an “unfolding level” N = 4. This builtin can be defined in Prolog as follows (but can, of course, be implemented more efficiently at a low level):

```
skip(L,N,LS,NS) :- skip(L,N,LS,NS,0).


skip(LS,0,LS,NS,NS) :- !.
skip([],_,[],NS,NS).
skip([_|Ls],N,LRs,Ns0,Ns) :-
        N1 is N-1,
        Ns1 is Ns+1,
        skip(Ls,N1,LRs,Ns0,Ns1).
```

We now return to our original program and make use of the proposed builtin (note that the “flattening parameter” N can be now chosen dynamically):

```
vproc_opt([],[],0).
vproc_opt(L,LR,N) :-
        N > 0,
        skip(L,N,LS,NS),
        skip(LR,NS,LRS,NS),
        vproc_opt(LS,LRS,NS) & vproc_opt_n(NS,L,LR).
```

```
vproc_opt_n(0,_,_).
vproc_opt_n(N,[L|Ls],[LR|LRs]) :-
        N > 0,
        N1 is N-1,
        vproc_opt_n(N1,Ls,LRs) & process_element(L,LR).
```

We have included the skip/4 predicate as a C builtin in the &–Prolog system and run the above program. The result is shown in Figure 4.7. The large delays are due to the traversal of the list made by skip/4. Note, however, how the tasks are created in groups of four corresponding to the dynamically selected increment, which can now be made arbitrarily large. We believe that this idea would also be useful when implemented at an even lower level [HC96].

It is worth noting that, in this case, the predicate skip/4 not only returns pointers to sublists of a given list, but is also able to construct a new list composed with free variables. This allows spawning independent parallel processes, each one of them working in separate segments of a list. This, in some sense, mimics the so–called *poslist* and *neglist* identified in the Reform Compilation at run–time. Though this solution gives, obviously, poorer performance than a compile–time approach.

Note also that other builtins similar to skip could be proposed for other types of data structures and for each type of traversal allowed by each of those data structures.

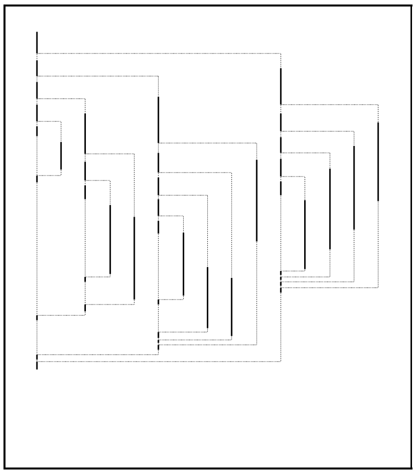


Figure 4.9: Vector operation, list prebuilt (10 el./8 proc.)

As an example, we may want the splitting of the list to be used afterwards (for example, because it is needed in some further similar processing). We can use the skip/4 predicate to build a skiplist/3 predicate as follows:

```
skiplist([], _N, []):- !.
skiplist(L, N, [L|LSs]):-
        skip(L, N, LS, _M),
        skiplist(LS, N, LSs).
```

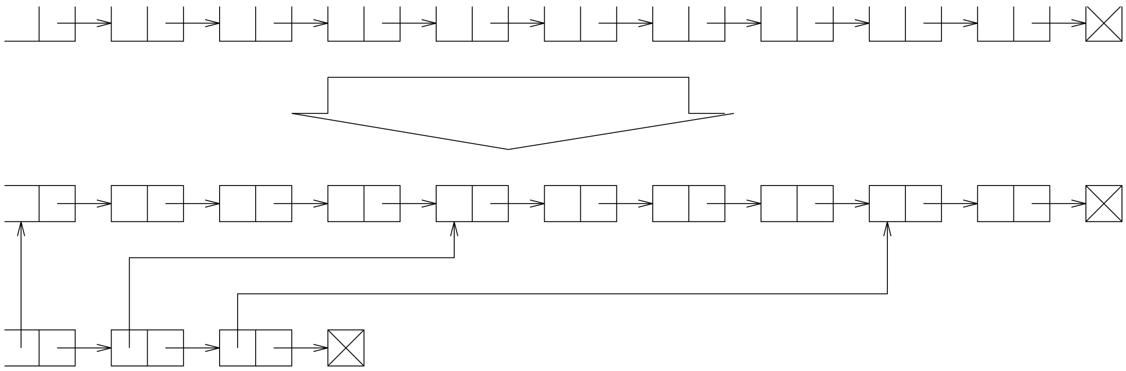


Figure 4.10: “Skiplist” operation, 10 elements in 4

A typical call to `skiplist/3` would be done with the two first arguments instantiated; the third argument would return pointers to sublists of the first argument or, under a more logical point of view, the third argument describes a set of sublists of the first argument by means of difference lists. Figure 4.10 depicts this situation, and Figure 4.9 shows the result of an execution where the input and output data has been pre-processed using this predicate. This list preprocessing does not appear in Figure 4.9, as an example of the reuse of a previously traversed list.

4.2.6 Performance Evaluation

In order to assess the relative performance of the various techniques discussed, we have run the examples on a larger (240 elements) list. The results presented in Table 4.1 show the corresponding execution times. The column *Relative Speedup* refers to the speedup with respect to the parallel execution in one processor, and the column *Absolute Speedup* measures the execution speed with respect to the sequential execution. The numbers between parentheses to the right of some benchmark names represent the skipping factor chosen.

Overheads associated with scheduling, preparing tasks for parallel execution, etc. make the parallel execution in one processor be slower than the sequential execution. This difference is more acute in very small grained benchmarks, as the one we are dealing with.

The speedups suggested by Figures 4.4 to 4.9 may not correspond with those in the table — the length of benchmark run and the skip/unfolding increment chosen in the two cases is different, and so is the distribution of the tasks (the skip increment for the benchmarks in Table 4.1 is given between parentheses). In fact, some figures suggest a slowdown where the table shows a speedup. On the other hand, this indicates that processing larger lists can take more advantage from the proposed techniques, because the relative overhead from traversing the list is comparatively less.

Method	**Time (ms)**	**Relative Speedup**	**Absolute Speedup**
Sequential	127	—	1
Parallel, 1 processor	153	1	0.83
Giving away recursion	134	1.14	0.94
Keeping recursion	41	3.73	3.09
Skipping (8)	30	5.1	4.23
Skipping (30)	28.5	5.36	4.45
Pre–built skipping list (8)	28	5.4	4.53
Pre–built skipping list (30)	26.5	5.77	4.79
Reform Compilation (8)	27	5.6	4.7
Data Parallel	26	5.88	4.88

Table 4.1: Times and speedups for different list access, 8 processors.

It can also be noted how a pre–built skipping list with a properly chosen increment beats the reformed program. Of course a reformed program with the same unfolding level would, in principle, at least equal the program with the pre–built list. But the point is that the reformed program was statically transformed, whereas the skiplist version can change dynamically, and be useful in cases where the same data is used several times in the same program.

4.3 Constant Time Access Arrays in Prolog?

Finally, and for the sake of argument, we propose a simple-minded approach to the original problem using the real "arrays" in standard Prolog, i.e. terms. Of course the use of this technique is limited by the fact that *term arity is limited in many Prolog implementations*, but this could be very easily cured. In the query we create a vector of length `N` using `functor/3`, initialize it, and then pass it on to a "vector" version of `vproc` (we could, of

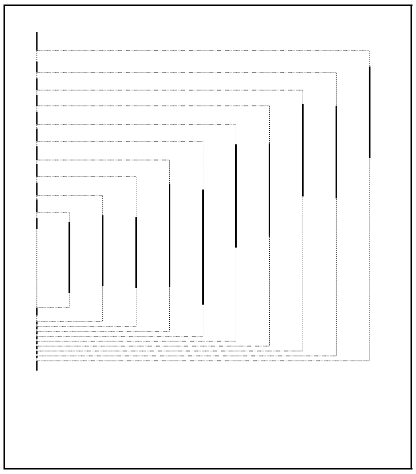


Figure 4.11: Vector operation, constant access arrays (10 el./8 proc.)

course, also start with a list, as in previous examples, and convert it into a vector before calling the parallelized "vector" version of `vproc`):

```
vproc(0,_,_).
vproc(_,V,VR) :-
        I>0,
        I1 is I-1,
        vproc(I1,V,VR) & process_element(I,V,VR).
```

Element access is done in constant time using `arg/3`:

```
process_element(I,V,VR) :-
        arg(I,V,H),
        HR is ((((H * 2) / 5)^2)+(((H * 6) / 2)^3))/2,
        arg(I,VR,HR).
```

The results are presented in Figure 4.11. In this example we are using a simple minded loop which creates tasks recursively, but the same techniques illustrated in previous examples could be applied to this "real array" version: it is easy now to modify the above program as in the previous examples in order to create the tasks in groups of N, but now without having to previously traverse the data structure, as was the case when using the `skip` builtin.

The result appears in Figure 4.12. From this figure it may seem that there is no performance improvement derived from using this strategy. This is due to the fact that the execution depicted is very small, and the added overhead of calculating the "splitting point" becomes a sizeable part of the whole execution. As in Table 4.1, in Table 4.2 larger lists and skipping factors were chosen, achieving better speedups than the simple parallel scheme. Since no real traversal is needed using this representation, the amount of items traversed can be dynamically adjusted with no extra cost.

Method	Time (ms)	Relative Speedup	Absolute Speedup
Sequential	149	—	1
Parallel, 1 processor	174	1	0.85
Keeping recursion	45	3.8	3.31
Binary startup	38	4.5	3.92
Skipping (8)	31.2	5.57	4.77
Skipping (30)	29.5	5.89	5.05

Table 4.2: Times and speedups for vector accesses

A more even load distribution than that obtained with the simple recursion scheme can be achieved by using a binary split. This is equivalent to dynamically choosing the splitting step to be half the length of the sub–vector assigned to the task. Figure 4.13 depicts this scheme. As in Figure 4.12, the comparatively large overhead associated with the determination of the splitting point makes this execution appear larger than that corresponding to the simple recursive case. But again, Table 4.2 reflects that for large enough executions, its performance can be placed between the simple recursion scheme and a carefully chosen skipping scheme.

It is clearly also trivial to convert from a list representation to a "vector representation" — e.g. for the one dimension case:

```
vectorize(L,V) :- vectorize(L,0,V) .

vectorize([],N,V) :-
        functor(V,storage,N).
vectorize([H|T],N,V) :-
        N1 is N+1,
        vectorize(T,N1,V),
        arg(N1,V,H).
```

Comparing Tables 4.1 and 4.2 some conclusions can be drawn. First, the structure–based programs are slightly slower than their list–based counterparts. This is understand-

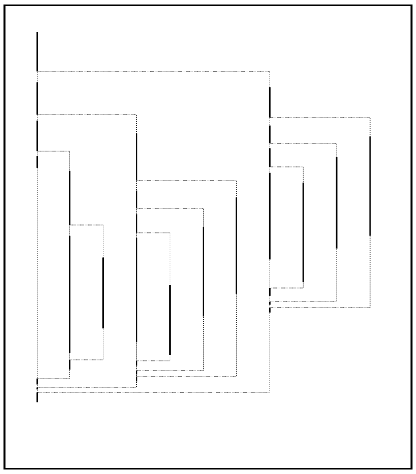


Figure 4.12: Vector operation, constant time access arrays, skipping, 10 el./8 proc.

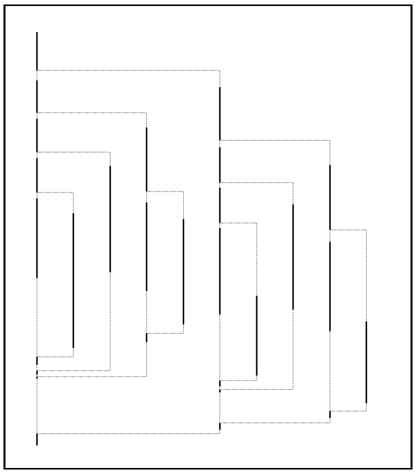


Figure 4.13: Vector operation, constant time access arrays, binary startup, 10 el./8 proc.

able in that using structures as arrays involves an index handling that is less efficient (or, rather, that has been less optimized) than in the case of lists. But the fact that accessing any element in a structure is, in principle, a constant–time operation, allows a comparatively efficient implementation of the dynamic *skip* strategy. This is apparent in that the speedups attained with the arrays version of the skipping technique are better than those corresponding to the list–based programs. The absolute speed is less; this can be attributed to the fact that the &–Prolog version with which these times were taken has the `arg/3` builtin written in C, with the associated overhead of calling and returning from a C function. This could be improved making `arg/3` (or a similar primitive) a faster, WAM–level instruction. Again, if we want (or have to) use lists, a low–level `vectorize/2` builtin could be fast enough to translate a list into a structure and still save time with respect to a list–based implementation processing the resulting structure in a divide–and–conquer fashion.

Finally, following on on this idea, we illustrate how one could even build a quite general purpose "FORTRAN-like" constant access array library without ever departing from standard Prolog or, eliminating the use of "`setarg`", even from "clean" Prolog. It is not that we are supporting the use of these data structures, but rather we are simply trying to make the point that if one really, really, wants them, then the arrays are there. The solution we propose is related to the standard "logarithmic access time" extensible array library written by D.H.D.Warren. In this case, we obtain constant (rather than logarithmic) access time, with the drawback that arrays are, at least in principle, fixed size.

We begin by defining the "type" array. Essentially, an array is a term of arity two which contains as its first argument a list of integers which correspond to the dimensions of the array (thus we can have arrays of arbitrary dimensions) and as its second argument a term whose arity is the total number of cells in the array (and thus represents the total amount of storage needed by the array):

```
is_array(matrix(D,S)) :-
        functor(S,storage,L),
        multiply_list(D,L).

multiply_list([],1).
multiply_list([I|Is],N) :-
        multiply_list(Is,N1),
        N is N1 * I.
```

Arrays can be created, in full FORTRAN tradition, by performing a call to `dimension/2`, where the first argument is a list with the dimensions of the array and the second argument returns the array:

```
dimension(D,matrix(D,S)) :-
        multiply_list(D,Nelements),
        functor(S,storage,Nelements).
```

Note, however, that with judicious use of delays (or in a CLP language) one can also create arrays through a simple call to the type definition predicate.

All elements of the "storage" part are accessible as arguments of a structure in time proportional to the number of dimensions of the matrix:

```
access(matrix(D,S),I,X) :-
        compute_offset(I,D,Offset),
        arg(Offset,S,X).

compute_offset([I],[D],I) :-
        I>0, I=<D, !.
compute_offset([I|Is],[D|Ds],Offset) :-
        I>0, I=<D, !,
        compute_offset(Is,Ds,Offset1),
        I1 is I-1,
```

```
        Offset is D * I1 + Offset1.
compute_offset(_,_,_) :-
        format("Warning: access out of bounds in array.",[]).
```

Finally, if one really, really wants to have everything one has in FORTRAN, then even destructive assignment is available:

```
setel(matrix(D,S),I,X) :-
        compute_offset(I,D,Offset),
        setarg(Offset,S,X).
```

However, one would hope that compilation technology would make the need for resorting to these extremes unnecessary.

The definitions above are meant as a description of the logical meaning of the operations on arrays (except for the destructive assignment, of course). From a practical point of view, these definitions should at least be changed to compute with an accumulating parameter. Also, use of delay (or CLP) can make them fully reversible. More realistically, all these operations should be builtins (or, even better, native instructions) for performance reasons. Note that calls to dimension, access, set, etc. could in any case often be very efficiently compiled in-line to a specialized call to `functor`, `arg`, etc.

4.4 A Transformation to Obtain More Parallelism

Section 4.3 introduced and evaluated a scheme for the parallel execution of data-parallel programs, in which the *splitting point* of the computation was set dynamically according to the size of the data. This splitting point was chosen aiming at a better balance of the work to be performed (including the creation of parallel tasks) between the two branches of parallel computation. In this section we will explore this issue more in depth, and we will develop a transformation, improving that in [DJ94], to cope with the cases where the amount of work needed to find the size of a data structure increases with this size. We will also discuss when this transformation can be applied, based on the properties of the operations to be performed on the components of the data structure.

The transformations we will propose are not likely to be fully automatic at the present stage of compilation techniques; however, we want to formalize a little bit the techniques we have evaluated experimentally so far in this chapter, and identify where they can be applied.

4.4.1 The Basic Binary Transformation

Some computations frequently found in data-parallel algorithms can be described as a transformation of an input $\overline{x}$, consisting of one or several actual arguments, which can be expressed as follows:

$$p(\overline{x}) = d(e^0(\overline{x})) \oplus d(e^1(\overline{x})) \oplus d(e^2(\overline{x})) \oplus \cdots d(e^{n-1}(\overline{x})) \oplus b(e^n(\overline{x})) \tag{4.1}$$

$e^i(\overline{x})$ obtains from the input data $\overline{x}$ the item needed in the ith step of the computation; e should meet the property $e^{i+1}(\overline{x}) = e^i(e(\overline{x}))$. The ith item is transformed by the function d, and the result of these expressions are combined using the operator $\oplus$. The base case of the computation is given by the function b. The number n stands for the size $\rho(\overline{x})$ of $\overline{x}$ [LGH95, BK96, DL90], and is non-negative for any $\overline{x}$. As an example, the factorial function

$$n! = n * (n-1) * (n-2) * \cdots * 2 * 1$$

can be easily expressed as

$$n! = d(e^0(n)) \oplus d(e^1(n)) \oplus d(e^2(n)) \oplus \cdots d(e^{(n-1)}(n)) \oplus b(e^n(n))$$

with $\oplus \equiv *$, $e(x) = x - 1$, $d(x) = x$, and $b(x) = 1$.

Throughout the remaining of this section we will assume that the input arguments are ground (as usually needed by data-parallel computations) and that the output arguments are free variables. In fact, this latter requirement is not needed: calls can be made with the arguments in output positions (partially) instantiated, as long as these calls do not fail.

Using a mixture of logic and functional notation, a skeleton of a generic predicate which performs this computation can be written (Figure 4.14) [DJ94]. The functions $\oplus$, d, e, and b are supposed to be appropriately evaluated.

$$p(\overline{x}, \overline{y}) :- \ a(\overline{x}),\ p(e(\overline{x}),\ \overline{y}_1),\ \overline{y} = d(\overline{x}) \oplus \overline{y}_1.$$
$$p(\overline{x}, \overline{y}) :- \ \neg a(\overline{x}),\ \overline{y} = b(\overline{x}).$$

Figure 4.14: A recursion scheme

$a(\overline{x})$ determines whether the base case has been reached or not, and behaves as if defined by

$$a(\overline{x}) \quad \leftrightarrow \quad \rho(\overline{x}) > 0$$

$p(\overline{x},\overline{y}) :-\ a(\overline{x}),\ q(\overline{x},1,\rho(\overline{x}),\overline{y}_1), \overline{y} = \overline{y}_1 \oplus b(e^{\rho(\overline{x})}(\overline{x})).$
$p(\overline{x},\overline{y}) :-\ \neg a(\overline{x}), \overline{y} = b(\overline{x}).$

$q(\overline{x},m,m,\overline{y}) :-\ \overline{y} = d(e^{m-1}(\overline{x})).$
$q(\overline{x},m,n,\overline{y}) :-\ m < n,\ m' = \lfloor \frac{m+n}{2} \rfloor,\ q(\overline{x},m,m',\overline{y}_1),\ q(\overline{x},m'+1,n,\overline{y}_2),\ \overline{y} = \overline{y}_1 \oplus \overline{y}_2.$

Figure 4.15: A transformation to improve parallelism

The procedure shown in Figure 4.14 has data dependencies among the recursive calls: for the result $\overline{y}$ to be worked out, the rest of the calls in the body must have been finished, which prevents its parallelization (as done in [BLM93b]). In [DJ94] a simple transformation (Figure 4.15) is shown which improves the independence of the computation, provided that the operator $\oplus$ is associative. The idea is to split the computation in halves and combine the result of each half; this needs the associativity of $\oplus$. In that case, the two calls to $q/4$ can be computed in separate branches (probably in parallel) and combined afterwards. The program in Figure 4.15 carries two counters, m and n, which locate exactly which part of the computation is being worked out. When a leaf of the binary tree is reached, a partial result is evaluated from the input data. This result is recursively propagated upwards, and combined with other results obtained in the same way.

For an input of size n, the simple recursive scheme applies n times the function d and $n+1$ times the function e. The binary scheme applies n times the function d, but e is applied $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ times. This difference comes from the fact that in the simple recursive case the output of $e^n(\overline{x})$ is used to calculate $e^{n+1}(\overline{x})$, and these partial results are handed down in successive invocations of the predicate. In the doubly recursive program, the leaves of the tree receive the initial input argument and apply e the necessary number of times. If applying e is costly, it might not even be worthwhile to execute in parallel the transformed program. Numerical loops do not suffer usually from this drawback, but, as we will see, algorithms using more complex data structures can be affected by it.

For example, the following simple recursive program accepts a list of numbers as input and returns as output another list composed by applying a (unevaluated) function `f` to each of them:

```
apply([], []).
apply([I|Is], [O|Os]):-
```

```
        O = f(I),
        apply(Is, Os).
```

This predicate can be transformed using the scheme in Figure 4.14 by establishing the following mapping:

Schematic operation	Concrete operation
$\rho(\overline{x}) = n$	`length(X, N)`
$a(\overline{x})$	`X = [_|_]`
$e(\overline{x}) = \overline{y}$	`X = [_|Y]`
$e^m(\overline{x}) = \overline{y}$	`nthtail(M, X, Y)`
$d(\overline{x}) = \overline{y}$	`X = [W|_], Y = [f(W)]`
$\overline{x} \oplus \overline{y} = \overline{z}$	`append(X, Y, Z)`
$b(\overline{x}) = \overline{y}$	`Y = []`

The `append/3` predicate is associative if it is viewed as a function with the first and second arguments as input. Since the type of associative operators is the same for both input and output, the function d has to return a list. The transformed program is as follows:

```
apply(X, Y):- \+ X = [_|_], Y = [].
apply(X, Y):-
        X = [_|_],
        X = [_|X1],
        apply(X1, Y1),
        X = [W|_],
        Y2 = [f(W)],
        append(Y2, Y1, Y).
```

which, after some sensible local rewriting (which may be performed by partial evaluators [DGT96, PH95, Sah90]), yields the following Prolog program:

```
apply([], []).
apply([X|Xs], Y):-
        apply(Xs, Ys),
        append([f(X)], Ys, Y).
```

The binary transformation applied to this program produces (again, after some sensible rewriting which groups and reorders the unifications) the following code:

```
apply_bin([], []).
apply_bin([X|Xs], Y):-
        length([X|Xs], L),
        apply_q([X|Xs], 1, L, Y).

apply_q(X, M, M, [f(W)]):-
        M1 is M - 1,
        nthtail(M1, X, [W|_]).
apply_q(X, M, N, Y):-
        M < N,
        Mp is (M + N) // 2,
        apply_q(X, M, Mp, Y1),
        M1 is Mp + 1,
        apply_q(X, M1, N, Y2),
        append(Y1, Y2, Y).
```

`nthtail/3` is supposed to return in the third argument the *nth* tail of a given list (being the 0-th tail the initial list itself; thus, `nthtail/3` meets the conditions needed to model correctly e). It is possible to make `append/3` in the last clause to execute in constant time by using difference lists, but unfortunately `length/2` and `nthtail/3` cannot be changed in a similar way. It is clear that much of the advantage of the parallel execution can be lost in traversing lists; but if d is costly to execute, this traversal might be worthwhile.

The bottom line is that if evaluating e^{n+1} is harder than evaluating e^n, as in this case, the doubly recursive program has a higher complexity than the original program, which, depending on how difficult is the evaluation of d compared with that of e, can render parallel execution worthless. On the other hand, programs where e^n has a constant cost regardless of the value of n clearly benefit from this transformation, as shown in [DJ94].

4.4.2 An alternative transformation

The overhead of the repeated calls to e can be reduced by passing down partial results of the application of the function e to $\overline{x}$, as in Figure 4.16.

The main difference with the scheme in Figure 4.15 is that the function e is applied incrementally to the branches of the computation, instead of being called only at the leaves. Since the basic case function b is applied to the value $e^{\rho(\overline{x})}(\overline{x})$, and this value is incrementally calculated along the branches, we need to distinguish the left and right

$$p(\overline{x},\overline{y}) :- a(\overline{x}),\ q_r(\overline{x},\rho(\overline{x}),\overline{y}).$$
$$p(\overline{x},\overline{y}) :- \neg a(\overline{x}),\ \overline{y} = b(\overline{x}).$$

$$q_r(\overline{x},n,\overline{y}) :- n = 1,\ \overline{y} = d(\overline{x}) \oplus b(e(\overline{x})).$$
$$q_r(\overline{x},n,\overline{y}) :- n > 1,\ m = \lfloor \tfrac{n}{2} \rfloor,\ q_l(\overline{x},m,\overline{y}_1),\ q_r(e^m(\overline{x}),n-m,\overline{y}_2),\ \overline{y} = \overline{y}_1 \oplus \overline{y}_2.$$

$$q_l(\overline{x},n,\overline{y}) :- n = 1,\ \overline{y} = d(\overline{x}).$$
$$q_l(\overline{x},n,\overline{y}) :- n > 1,\ m = \lfloor \tfrac{n}{2} \rfloor,\ q_l(\overline{x},m,\overline{y}_1),\ q_l(e^m(\overline{x}),n-m,\overline{y}_2),\ \overline{y} = \overline{y}_1 \oplus \overline{y}_2.$$

Figure 4.16: Handing down applications in a binarized function

branches of the computation, being the rightmost branch the one which eventually applies b. The transformation in Figure 4.16 can be applied to the `apply/2` program to yield the following code (again, after some sensible rewriting):

```
apply_bin_pd([], []).
apply_bin_pd(X, Y):-
        length(X, Lx),
        apply_bin_pd_r(X, Lx, Y).

apply_bin_pd_r([X], 1, [f(X)]).
apply_bin_pd_r(X, N, Y):-
        N > 1,
        M is N // 2,
        apply_bin_pd_l(X, M, Y1),
        nthtail(M, X, Xm),
        M1 is N - M,
        apply_bin_pd_r(Xm, M1, Y2),
        append(Y1, Y2, Y).

apply_bin_pd_l([X|_], 1, [f(X)]).
apply_bin_pd_l(X, N, Y):-
        N > 1,
        M is N // 2,
        apply_bin_pd_l(X, M, Y1),
        nthtail(M, X, Xm),
        M1 is N - M,
        apply_bin_pd_l(Xm, M1, Y2),
```

```
append(Y1, Y2, Y).
```

The function d is applied the same number of times as in the scheme of Figure 4.15, but e is called less times. The number of applications of e is given by $f(\rho(\overline{x}))$, where f is defined as

$$f(n) = \begin{cases} 0 & \text{if n = 0} \\ g(n) + 1 & \text{if n > 0} \end{cases}$$

$$g(n) = \begin{cases} 0 & \text{if n = 1} \\ \lfloor \frac{n}{2} \rfloor + g(\lfloor \frac{n}{2} \rfloor) + g(n - \lfloor \frac{n}{2} \rfloor) & \text{if n > 1} \end{cases}$$

This can be approximated by the upper bound $\Theta_f(n) = \lfloor \frac{n \log_2 n}{2} \rfloor$, which is accurate when n is a power of two.

4.4.3 A More General Transformation

The previous transformations rely on having an (associative) operation which combines the different partial results to yield the final result. Knowing that the operator was meant to be a function made it easy to distinguish which arguments were "input" and which were "output".

Some programs are difficult, or at least unnatural, to fit in the function-based scheme. This is the case of programs using partially instantiated structures, or with several output arguments. For this type of programs, the requirement of having an associative function combining partial results can be translated into having a predicate meeting certain properties, which will allow the transformation into the doubly recursive form.

There are other problems related with the use of certain data structures: indexed data structures are specially useful in data-parallel programs, and mapping those structures to the templates shown before yields programs with an odd appearance. Figure 4.17 presents one example: every *nth* element in the input structure is mapped onto the *nth* argument of the output structure (the application of the function `f` to an element `N` of the input vector `I` is, for clarity, encapsulated inside the predicate `set_element/3`). Since any element in the structure can be accessed in constant time and each of these elements is transformed independently of the rest, the use of data parallelism and of the proposed transformation techniques is appealing. But every application of `f` (the corresponding to d) is performed on a different element of the input structure, while in previous examples the element to which d should be applied was generated directly by transforming the initial argument. In the current case, the input argument needs to identify which element in the structure has to be accessed: this can be accomplished, for example, by augmenting this argument to be a tuple $\langle$*index*, *structure*$\rangle$, so that the index of the argument being accessed at every computation step is encapsulated with the data. Since the left and right operands of an associative operator have the same type, the output argument must be a similar tuple. In Figure 4.18 the arguments have been explicitly grouped in tuples; the following table shows how each part of the Prolog program is mapped onto the translation scheme:

Abstract Operation	**Concrete Operation**
$\rho(\overline{x}) = n$	`X = (_, V), length(V, N)`
$a(\overline{x})$	`X = (N, V), N > 0`
$e(\overline{x}) = \overline{y}$	`(N, X) = (N-1, Y)`
$e^m(\overline{x}) = \overline{y}$	`(N, X) = (N-M, Y)`
$d(\overline{x}) = \overline{y}$	`apply_to_element(X, Y)`
$\overline{x} \oplus \overline{y} = \overline{z}$	`X = Y, X = Z`
$b(\overline{x}) = \overline{y}$	`Y = _`

The composition of partial results using $\oplus$ needs to work on independent data structures (otherwise, the parallelization would not be possible). This forces `apply_to_element/2` to create a new output argument—which, in this case, is a whole new vector with only an argument instantiated. The $\oplus$-composition of the intermediate results boils down to unification. This is not what one usually wants: the standard approach of the programmer would be to have just one output argument and instantiate its contents during the computation. But this would, in principle, break the conditions for strict independence, since free variables would be shared by parallel goals—although no two parallel goals would try to instantiate the same variable.

The introduction of indices into the simple recursive scheme, and the presence of shared logical variables will force us to consider the next and, for now, last scheme for predicate binarization.

4.4.4 Indices Made Explicit

Figure 4.18 shows a program which recursively splits a vector-to-vector computation in halves, so that the overhead of starting up parallel tasks is distributed among processors.

```
ar_apply(I, O):-
        functor(I, Name, N),
        functor(O, Name, N),
        ar_apply(N, I, O).

ar_apply(0, _, _).
ar_apply(N, I, O):-
        N > 0,
        N1 is N - 1,
        set_element(N, I, O),
        ar_apply(N1, I, O).

set_element(N, I, O):-
        arg(N, I, ArgN),
        arg(N, O, f(ArgN)).
```

Figure 4.17: `apply` program with structures

```
ar_apply(I, O):-
        functor(I, Name, N),
        functor(O, Name, N),
        ar_apply_tu((N, I), (N, O)).

ar_apply_tu(X, _Y):-
        X = (N, _V), N = 0.
ar_apply_tu(X, Y):-
        X = (Ni, I), Ni > 0,
        Ni1 is Ni - 1,
        apply_to_element(X, Xap),
        Xnew = (Ni1, I),
        ar_apply_tu(Xnew, Ynew),
        compose(Xap, Ynew, Y).

apply_to_element((Nx,X), (_Ny,Y)):-
        functor(X, Name, Ar),
        functor(Y, Name, Ar),
        arg(Nx, X, H),
        NewH = f(H),
        arg(Nx, Y, NewH).

compose(X, X, X).
```

Figure 4.18: `apply` program with structures and I/O arguments

A problem in this code is the replication of the index in the input and output arguments and the creation of a new vector for each element, when only an element of this new vector is bound. Many mapping operations use the same index for input and output arguments, which means that in similar data structures, an information item in the input structure is transformed into another information item in the corresponding position of the output structure, and so there is no need to have separate input and output indices. It is even possible that the index for the output argument can be computed on the fly, if a different location is required[4]. It may be then a good idea to make an index explicit in the translation scheme, which can be accomplished by augmenting the expression 4.1 to become

$$p(\overline{x}) = d(e^0(\overline{x}), 0) \oplus d(e^1(\overline{x}), 1) \oplus d(e^2(\overline{x}), 2) \oplus \cdots d(e^{n-1}(\overline{x}), n-1) \oplus b(e^n(\overline{x}), n) \quad (4.2)$$

where d has now an additional argument stating which iteration step it belongs to. Following on with the naming convention we have been using so far, we can develop a variant (Figure 4.19) of the recursion scheme in 4.14 which implements the above computation.

$$
\begin{array}{l}
p(\overline{x}, \overline{y}) :- \ a(\overline{x}),\ \overline{y} = b(\overline{x}). \\
p(\overline{x}, \overline{y}) :- \ \neg a(\overline{x}),\ q(\overline{x}, 1, \overline{y}). \\
\\
q(\overline{x}, m, \overline{y}) :- \ m = \rho(\overline{x}),\ \overline{y} = d(\overline{x}, m) \oplus b(e(\overline{x}), m+1). \\
q(\overline{x}, m, \overline{y}) :- \ m < \rho(\overline{x}),\ q(e(\overline{x}), m+1, \overline{y}_1),\ \overline{y} = d(\overline{x}, m) \oplus \overline{y}_1.
\end{array}
$$

Figure 4.19: A recursion scheme including indices

$$
\begin{array}{l}
p(\overline{x}, \overline{y}) :- \ \neg a(\overline{x}),\ \overline{y} = b(\overline{x}). \\
p(\overline{x}, \overline{y}) :- \ a(\overline{x}),\ q_r(\overline{x}, 1, \rho(\overline{x}), \overline{y}). \\
\\
q_r(\overline{x}, m, m, \overline{y}) :- \ \overline{y} = d(\overline{x}, m) \oplus b(e(\overline{x})). \\
q_r(\overline{x}, m, n, \overline{y}) :- \ m < n,\ n_1 = \lfloor \frac{m+n}{2} \rfloor,\ q_l(\overline{x}, m, n_1, \overline{y}_1), \\
\qquad\qquad m_1 = n_1 + 1,\ q_r(e^{m_1 - m}(\overline{x}), m_1, n, \overline{y}_2),\ \overline{y} = \overline{y}_1 \oplus \overline{y}_2. \\
\\
q_l(\overline{x}, m, m, \overline{y}) :- \ \overline{y} = d(\overline{x}, m). \\
q_l(\overline{x}, m, n, \overline{y}) :- \ m < n,\ n_1 = \lfloor \frac{m+n}{2} \rfloor,\ q_l(\overline{x}, m, n_1, \overline{y}_1), \\
\qquad\qquad m_1 = n_1 + 1,\ q_l(e^{m_1 - m}(\overline{x}), m_1, n, \overline{y}_2),\ \overline{y} = \overline{y}_1 \oplus \overline{y}_2.
\end{array}
$$

Figure 4.20: Handing down applications with explicit indices

Using a scheme similar to that of Figure 4.16 a recursion pattern can be written which takes explicitly into account the presence of indices pointing to items in data structures (Figure 4.20). In this code, the two indices represent the *window* each subcomputation is in charge of. When both are equal, there is only one element in the window to be processed, and its index is readily available.

[4]But we will not take this possibility into account in the transformation we will develop now, for the sake of clarity.

Scheme	Concrete
$a(\overline{x})$	`atom(X)`
$b(\overline{x}, \overline{y})$	`Y = 0`
$d(\overline{x}, m, \overline{y})$	`arg(M, X, Y)`
$e^m(\overline{x}, \overline{y})$	`X = Y`
$\oplus(\overline{y}_1, \overline{y}_2, \overline{y})$	`Y is Y1 + Y2`
$\rho(\overline{x}, l)$	`functor(X, _, L)`

Table 4.3: Adding elements

Scheme	Concrete
$a(\overline{x})$	`atom(X)`
$b(\overline{x}, \overline{y})$	`Y = []`
$d(\overline{x}, m, \overline{y})$	`arg(M, X, A), Y = [A]`
$e^m(\overline{x}, \overline{y})$	`X = Y`
$\oplus(\overline{y}_1, \overline{y}_2, \overline{y})$	`append(Y1, Y2, Y)`
$\rho(\overline{x}, l)$	`functor(X, _, L)`

Table 4.4: Mapping a structure onto a list

Figures 4.21 and 4.22 show two examples of the application of this skeleton to two programs: the first one adds the (numerical) arguments of a structure, and the second one creates a list containing all the elements in a structure (similar to the `=../2` Prolog predicate, but without including the functor name). They where written by applying the definitions in Tables 4.3 and 4.4 to the skeleton in Figure 4.20, and performing (by hand) some code cleaning (inlining the definition of the predicates, propagating the local unifications, and simplifying obvious arithmetical operations). It is to be noted that, in the case of Table 4.4, the list-related operations could have used difference lists to achieve a better performance.

This transformation, applied to the program in Figure 4.18 which maps a Prolog structure into another structure, yields a code (Figure 4.23) similar to that used in Section 4.3 to evaluate the doubly-recursive parallel version whose execution was depicted in Figure 4.13. The $\oplus/3$ predicate performs just unification, but the application of $d(\overline{x}, m)$ creates a new data structure on the fly, with the associated overhead, even if only a single element of this data structure is actually accessed. This program can be systematically simplified by inlining the definition of $\oplus/3$ and by lifting the calls to `functor/3` in `apply_to_element/3` to the toplevel predicate; this is possible because, after the propagation of the unifications performed by $\oplus/3$, all the calls to `functor/3` can be shown to be applied to the same parameters.

The safeness of the parallel execution with this last (logically correct) simplification is challenged. When every recursive call returned a new data structure with fresh variables, which was combined afterwards, the independence of the computations was ensured; however, when the calls to be executed in parallel share variables, they can be, at most, non strictly independent. This means that no two parallel goals should bind the same variable, although they have access to it. Our transformation ensures that dif-

```
p(I, 0):-
        atom(I).
p(I, O):-
        \+ atom(I),
        functor(I, _Name, Length),
        q_r(I, 1, Length, O).

q_r(Base, Indx, Indx, Element):-
        arg(Indx, Base, Element).
q_r(I, M, N, O):-
        M < N,
        N1 is (M + N) // 2,
        M1 is N1 + 1,
        q_l(I, M, N1, Ol),
        q_r(I, M1, N, Or),
        O is Ol + Or.

q_l(Base, Indx, Indx, Element):-
        arg(Indx, Base, Element).
q_l(I, M, N, O):-
        M < N,
        N1 is (M + N) // 2,
        M1 is N1 + 1,
        q_l(I, M, N1, Ol),
        q_l(I, M1, N, Or),
        O is Ol + Or.
```

Figure 4.21: Adding elements of a structure

```
p(I, []):-
        atom(I).
p(I, O):-
        \+ atom(I),
        functor(I, _Name, Length),
        q_r(I, 1, Length, O).

q_r(Base, Indx, Indx, ResOp):-
        arg(Indx, Base, Element),
        append([Element], [], ResOp).
q_r(I, M, N, O):-
        M < N,
        N1 is (M + N) // 2,
        M1 is N1 + 1,
        q_l(I, M, N1, Ol),
        q_r(I, M1, N, Or),
        append(Ol, Or, O).

q_l(Base, Indx, Indx, [ResOp]):-
        arg(Indx, Base, ResOp).
q_l(I, M, N, O):-
        M < N,
        N1 is (M + N) // 2,
        M1 is N1 + 1,
        q_l(I, M, N1, Ol),
        q_l(I, M1, N, Or),
        append(Ol, Or, O).
```

Figure 4.22: Mapping a structure onto a list

ferent calls to d are addressed to different indices in the data structure. Since different indices correspond to actual different elements, the non-strict independence is met, and the simplifications performed are safe with respect to correct parallel execution.

A last step in simplification comes from the observation that in Figure 4.24 the clauses

```
ar_apply_bin(X, Y):-
        atom(X),
        X = Y.
ar_apply_bin(X, Y):-
        \+ atom(X),
        functor(X, _Name, N),
        ar_apply_tu_r(X, 1, N, Y).

ar_apply_tu_r(X, M, M, Y):-
        apply_to_element(X, M, Y1),
        compose(Y1, _, Y).
ar_apply_tu_r(X, M, N, Y):-
        M < N,
        N1 is (M + N) // 2,
        ar_apply_tu_l(X, M, N1, Y1),
        M1 is N1 + 1,
        ar_apply_tu_r(X, M1, N, Y2),
        compose(Y1, Y2, Y).

ar_apply_tu_l(X, M, M, Y):-
        apply_to_element(X, M, Y).
ar_apply_tu_l(X, M, N, Y):-
        M < N,
        N1 is (M + N) // 2,
        ar_apply_tu_l(X, M, N1, Y1),
        M1 is N1 + 1,
        ar_apply_tu_l(X, M1, N, Y2),
        compose(Y1, Y2, Y).

apply_to_element(X, N, Y):-
        functor(X, Name, Arity),
        functor(Y, Name, Arity),
        arg(N, X, H),
        NewH = f(H),
        arg(N, Y, NewH).

compose(X, X, X).
```

Figure 4.23: A doubly recursive predicate

```
ar_apply_bin(X, Y):-
        atom(X),
        X = Y.
ar_apply_bin(X, Y):-
        \+ atom(X),
        functor(X, Name, N),
        functor(Y, Name, N),
        ar_apply_tu_r(X, 1, N, Y).

ar_apply_tu_r(X, M, M, Y):-
        apply_to_element(X, M, Y).
ar_apply_tu_r(X, M, N, Y):-
        M < N,
        N1 is (M + N) // 2,
        ar_apply_tu_l(X, M, N1, Y),
        M1 is N1 + 1,
        ar_apply_tu_r(X, M1, N, Y).

ar_apply_tu_l(X, M, M, Y):-
        apply_to_element(X, M, Y).
ar_apply_tu_l(X, M, N, Y):-
        M < N,
        N1 is (M + N) // 2,
        ar_apply_tu_l(X, M, N1, Y),
        M1 is N1 + 1,
        ar_apply_tu_l(X, M1, N, Y).

apply_to_element(X, N, Y):-
        arg(N, X, H),
        arg(N, Y, f(H)).
```

Figure 4.24: After further simplification

```
ar_apply_bin(X, Y):-
        atom(X),
        X = Y.
ar_apply_bin(X, Y):-
        \+ atom(X),
        functor(X, Name, N),
        functor(Y, Name, N),
        ar_apply_tu(X, 1, N, Y).

ar_apply_tu(X, M, M, Y):-
        apply_to_element(X, M, Y).
ar_apply_tu(X, M, N, Y):-
        M < N,
        N1 is (M + N) // 2,
        ar_apply_tu(X, M, N1, Y),
        M1 is N1 + 1,
        ar_apply_tu(X, M1, N, Y).

apply_to_element(X, N, Y):-
        arg(N, X, H),
        arg(N, Y, f(H)).
```

Figure 4.25: Identical clauses can be collapsed

corresponding to the predicates `arr_apply_tu_r/4` and `arr_apply_tu_l/4` are identical, and can thus be collapsed into a single predicate. Figure 4.25 shows the result of this simplification, which is exactly the code which was used to generate the trace depicted in Figure 4.13 and the data in Table 4.2.

4.5 Conclusions

We have argued that data–parallelism and and–parallelism are not fundamentally different and that the advantages of both can be obtained within the same system. We have argued that the difference lies in two main issues: memory management and fast task startup. Having pointed to recent progress in memory management techniques in and–

$$
\begin{aligned}
&p(\overline{x},\overline{y}) :- \neg a(\overline{x}),\ b(\overline{x},\overline{y}).\\
&p(\overline{x},\overline{y}) :- a(\overline{x}),\ \rho(\overline{x},l),\ q_r(\overline{x},1,l,\overline{y}).\\[1em]
&q_r(\overline{x},m,m,\overline{y}) :- d(\overline{x},m,\overline{y}_1),\ e(\overline{x},1,\overline{x}_1),\ b(\overline{x}_1,\overline{y}_2),\ \oplus(\overline{y}_1,\overline{y}_2,\overline{y}).\\
&q_r(\overline{x},m,n,\overline{y}) :- m > n,\ n_1 = \lfloor \tfrac{m+n}{2} \rfloor,\ q_l(\overline{x},m,n_1,\overline{y}_1),\\
&\qquad\qquad m_1 = n_1 + 1,\ e(\overline{x},m_1 - m,\overline{x}_1),\ q_r(\overline{x}_1,m_1,n,\overline{y}_2),\ \oplus(\overline{y}_1,\overline{y}_2,\overline{y}).\\[1em]
&q_l(\overline{x},m,m,\overline{y}) :- \overline{y} = d(\overline{x},m).\\
&q_l(\overline{x},m,n,\overline{y}) :- m > n,\ n_1 = \lfloor \tfrac{m+n}{2} \rfloor,\ q_l(\overline{x},m,n_1,\overline{y}_1),\\
&\qquad\qquad m_1 = n_1 + 1,\ e(\overline{x},m_1 - m,\overline{x}_1),\ q_l(\overline{x}_1,m_1,n,\overline{y}_2),\ \oplus(\overline{y}_1,\overline{y}_2,\overline{y}).
\end{aligned}
$$

Figure 4.26: Explicit indices and logical variables

parallelism we have concentrated on the topic of fast task startup, discussing the relevant issues, and proposed a number of solutions, illustrating the point made through visualizations of actual parallel executions implementing the ideas proposed. In summary, we argue that both approaches can be easily reconciled, resulting in more powerful systems which can bring the performance benefits of data–parallelism with the generality of traditional and–parallel systems.

The examples shown focused on a type of computations characterized by iterations over data structures which, in our case, are lists or arrays; we argue that these cases are very frequent, and that paying attention to them would improve the performance of many programs. We have shown some transformation techniques leading to a better dynamic load distribution that can improve the speedups obtained in parallel executions. However, the overhead associated with this dynamic distribution is large in the case of lists; better speedups can be obtained using data structures with constant access time, in which choosing arbitrarily the splitting point does not impose any additional overhead.

In other computation schemata the iteration is performed over a numerical parameter; while not directly characterizable (or usually seen) as "data–parallelism", this type of iteration can also benefit from fast task startup techniques. This very interesting issue has been discussed by Debray and Jain [DJ94], and shown also to achieve significant speedups for that class of problems. The transformation proposed by them, while adequate for numerical computations, adds a significant overhead when applied to computations dealing with other data structures. We have extended this transformation to improve its performance in the cases where the cost of computating intermediate arguments is not independent of their sizes, and to the cases where the data structures are indexed. The simply recursive programs, once transformed, are identical to those used in the examples of data-parallel computations; thus, the transformations proposed can be used to effectively change the initial programs into doubly recursive ones.

Chapter 5

Some Optimizations for Independent And-parallel Systems

Chapter Summary

There are a series of sources of overhead in the And-parallel execution of Prolog programs. Many of them are due to the need of maintaining a backtracking semantics compatible with that existing in the sequential execution. There are, however, many situations in which it is unnecessary to preserve this behavior, and therefore it becames possible to avoid the associated overhead. We present the design and evaluation of some low level optimizations for and-parallel executions in this chapter. These optimizations are aimed at removing the need of taking into account the possibility of backtracking which is finally not needed.

5.1 Introduction

The high-level description given in Chapter 2 may suggest that building a RAP system is relatively straightforward. This is however not true in the least: the design and implementation of correct and efficient mechanismos for such a system is not without conceptual and technical challenges:

Logical vs. Physical: the Warren Abstract Machine has been carefully crafted to support efficient sequential execution of logic programs. In its design the WAM takes full advantage of the direct correspondence between logical structure of the execution and its physical layout in the abstract machine's data areas, as well of as the ordering in memory of these areas. This allows a considerably simpler execution of very complex operations—e.g. backtracking becomes very efficient, since a choice point on the stack is capable of completely identifying the execution state existing at its creation time.

This is not anymore true when dealing with unrestricted scheduling of goals[1] [SH94]: the computation can be arbitrarily spread on the stacks of different agents, and the physical order of computations on each stack can be completely different from the logical one (e.g. a subgoal $\mathcal{A}$ which appears to the right of a subgoal $\mathcal{B}$ may appear on the stack in the opposite order [Her86a]).

This lack of matching between logical and physical view of the execution creates considerable difficulties. Positioning on a choice point is not sufficient anymore to get a view of a state of the execution (as in the case of sequential computations). This correspondence has to be explicitly recreated, using additional data structures (in our case, the information contained in the markers).

Backtracking Policy: the backtracking semantics described earlier specifies *what* (and *when*) has to be done, but does not hints how backtracking on distributed computations can be implemented. Two approaches are feasible:

1. *Private Backtracking:* each agent is allowed to backtrack only over parts of the computation that are lying in its own stacks. This simplifies the memory management, but requires implementation of synchronization mechanisms to transfer the backtracking activity between agents.
2. *Public Backtracking:* each agent is allowed to backtrack on the stacks of other agents. This avoids the additional costs of communication between agents, but makes garbage collection and the overall memory organization more complex.

Both approaches are difficult to implement, and add an additional overload to the system, which has either to send appropriate messages in order to communicate the need of backtracking or split memory stacks in order to let other processor to use them.

Trail Management: one of the main problems in managing backtracking in an and-parallel system is detecting the parts of the trail stack that need to be unwound (i.e., detecting bindings that have to be removed to restore the proper computation state). A popular model used by, e.g., ACE [PGH95], is based on a *segmented view* of the stack, where a segment is defined by a section of the stack between two consecutive choicepoints. This makes trail management slightly less efficient than

[1] We use the term *unrestricted scheduling* to specify that no restrictions are imposed during scheduling on the selection of the next piece of work.

in traditional sequential implementations (two pointers, instead of one, pointing to the beginning and end of the relevant section of the trail, need to be saved in each choicepoint), but considerably simplifies both backtracking and management of or-parallelism.

Garbage Collection: the use of private backtracking allows recovery of a considerable amount of garbage "on-the-fly" during execution. Nevertheless, garbage collection remains more complicated, due to the lack of correspondence between logical order of backtracking and physical distribution of the computation. Parts of computation which are not on the top of the stack may be backtracked over first, leaving behind *holes* in the stacks (this is the so–called "garbage slot" problem) that need to be properly tagged and eventually recovered.

From the previous points we can conclude that a sizeable part of And-parallel execution will finally be devoted to consult and manage information related with goal/task synchronization and with the (internal) state of the execution, penalizing the execution of the user code. The main associated problems are, besides the difficulty of the implementation in itself, the waste of time (which diminishes the system efficiency) and of memory, needed in order to maintain data-structures to guarantee a correct parallel execution. A great deal of this overhead comes from having the execution segments interspread through several physical stacks, which forces traversing several memory zones in order to generate a comprehensive *view* of the execution structure, and to know where and to where backtracking must be performed.

The schemata seen so far are however designed for the most general case, and follow an approach which can satisfy all possible situations. Unsurprisingly, this "most general approach" is also very costly. Special properties of particular executions can be taken advantage of in order to simpilify the execution profile.

The code transformations proposed in Chapter 4, which help in obtaining more efficiency, do not change the number of parallel *user goals* generated.[2] Techniques such as granularity analysis, applicable at the Prolog code level, do reduce clearly this number (see, e.g., Figure 3.25). It possible to go to a lower implementation level and use properties such as determinism in order to allow partially the use of additional memory without reducing the number of parallel tasks. We will perform this by identifying and taking advantage of some program properties which allow reducing the overhead imposed by the general case of parallel execution, but without forgetting that this simplification must

[2] I.e., the number of parallel goals which perform work explicitly expressed by the programmer remains the same. More *control* tasks, aiming at a better load distribution, can however appear.

not invalidae the general case.

In this chapter we will describe two low level optimizatins which need some changes in the WAM implementation, and we will study its impact in the quality of the execution: the *last parallel call optimization* (LPCO) and the *backtracking families optimization*. The first one is completely based on the run time behavior of the program, while the second requires some informacion which, as goal independence, should be statically gathered before the execution.

5.2 Last Parallel Call Optimization

Last Parallel Call Optimization (LPCO) is a generalization of the Last Call Optimization [War80]—adopted in most sequential implementations—to the case of parallel calls. Its intent is to merge, whenever possible, distinct parallel conjunctions. Last Parallel Call Optimization can lead to a number of advantages (discussed later). The advantages of LPCO are very similar to those for last call optimization [War80] in the WAM. The conditions under which the LPCO applies are also very similar to those under which last call optimization is applicable in sequential systems.

Consider first the following example:

```
?- p & q.

p:- r & s.

q:- t & u.
```

The and-tree constructed is shown in Figure 5.1(i). One can reduce the number of parcall nodes, at least for this example, by rewriting the query[3] as

```
?- r & s & t & u.
```

Figure 5.1(ii) shows the and-tree that will be created if we apply this optimization. Note that executing the and-tree shown in Figure 5.1(ii) will require less space because the parcall frames for `r & s` and `t & u` will not be allocated. The single parcall frame allocated will have two extra goal slots compared to the parcall frame allocated for `p & q` in Figure 5.1(i). It is possible to detect cases such as the one above at compile time. However, our aim is to accomplish this saving in time and space at runtime. Thus, for the example above, our scheme will work as follows. When the parallel calls `r & s` and

[3] Under the assumption that the two clauses are the only matching ones.

`t & u` are made, the runtime system will recognize that they are the last parallel calls in their respective clauses and that the parallel call `p & q` is immediately above. Instead of allocating a new parcall frame some extra information will be added to the parcall frame of `p & q` and allocation of a new parcall frame avoided. Note that this is only possible if both `p` and `q` are determinate, i.e. they have at most one matching clause. The extra information added will consist of adding slots for the goals `r`, `s`, etc. In particular, no new control information needs to be recorded in the parcall frame of `p & q`. However, some control information, such as the number of slots, etc., need to be modified in the parcall frame of `p & q`. It is also necessary to slightly modify the structure of a slot in order to adapt it to the new pattern of execution.[4]

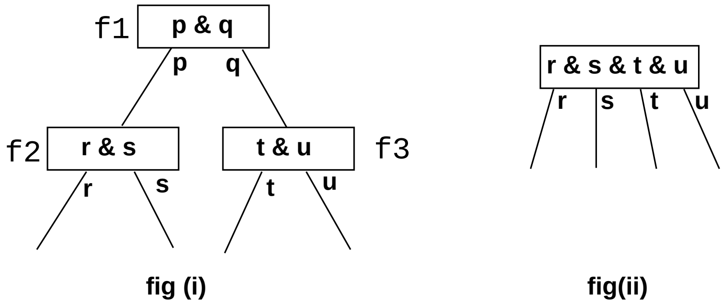


Figure 5.1: Optimization Schemes

It is important to observe that, if the goal `r` is to fail in inside mode, then in case (ii) (see Figure 5.1(ii)) killing of computation in sibling and-branches will be considerably simplified. In case (i) the failure will have to be propagated from parcall frame f2 to parcall frame f1. From f1 a kill message will have to be sent out to parcall frame f3. In case (ii) a linear scan of only one goal list is sufficient.

One could argue, as mentioned earlier, that the improved scheme described above can be accomplished simply through compile time transformations. However, in many cases this may not be possible. For example, if `p` and `q` are dynamic predicates or if there is not sufficient static information to detect the determinacy of `p` and `q`, then the compile-time analysis will not be able to detect the eventual applicability of the optimization. Our scheme will work even if `p` and `q` are dynamic or if determinacy information cannot be statically detected (because it is triggered only at runtime). Also, even more relevant, for many programs the number of parallel conjunctions that can be combined into one will only be determined at run-time [TPG94].

[4]For example, it is necessary to keep in each slot a pointer to the environment in which the execution of the corresponding subgoal will start.

In general, application of LPCO requires two *conditions* to be satisfied:

- determinacy of the computation between two nested parallel calls;
- non-existence of any continuation after the nested parallel calls (i.e., only the top-most parcall can have a continuation).

These conditions are satisfied by a large number of programs (e.g., tail-recursive programs) [GP95]. Work is in progress to generalize this optimization, so that it applies to a wider range of programs [PG94, PG95b]. It is important to observe that the cost of verifying applicability of LPCO at run-time is absolutely negligible (comparison of two pointers). This is a further justification for keeping the optimization as a pure run-time operation.

Executed goal	**LPCO execution**			
	fw/without *lpco*	fw/with *lpco*	bw/without *lpco*	bw/with *lpco*
bt_cluster	890	843 (5%)	929	853 (8%)
deriv(0)	94	34 (64%)	131	38 (71%)
poccur(5)	3216	3063 (5%)	3352	3226 (4%)
annotator(5)	1327	1282 (3%)	1334	1281 (4%)
matrix_mult(20)	1724	1649 (4%)	1905	1696 (11%)
search(1500)	2354	1952 (17%)	8370	2154 (74%)

Table 5.1: Timings with and without *lpco* (single processor)

Table 5.1 illustrates the results obtained executing some of the benchmarks using LPCO. The results are extremely good for programs with a certain structure. In particular, programs of the form `p(...) :- q(...) & p(...)`, where `q(...)` gives rise to a deterministic computation with a sufficiently deep level of recursion, will offer considerable improvement in performance.

Interesting results are also seen by examining the effect of *inside failures* during execution: the use of LPCO allows further improvement. The presence of a single parcall frame considerably reduces the delay of propagating kill signals (kill signals are sent to sibling and-branches by a failed subgoal in a parallel conjunction to remove them from the computation). Speedups of up to a 42% have been observed in a matrix multiplication.

5.3 Backtracking Families Optimization

LPCO uses characteristics of the program found out at run-time. However, it is interesting to explore whether similar ideas can be extended to cover more cases if some compile-time analysis information is available. In particular, we concentrate on the following types of information: knowledge that a goal (and its whole subtree) is deterministic, and knowledge that a goal has a single solution. In addition, knowledge that a goal will not fail is also quite useful [HR95]. It is beyond the scope of this paper to address how this information is gathered—the reader is referred to related work in the area of abstract interpretation based global analysis [JM94, BdlBH94c, SCWY91c, VD92, DLGH94]. We will address instead how such information can be exploited at the parallel abstract machine level.

We start by considering the case in which several parallel goals, perhaps not contiguous in the program text, but which are known to be deterministic, end up being executed on the same processor. As an example, consider the parallel call `a & b & c`, where `a`, `b` and `c` are known to be deterministic. If `a` and `b` are executed on the same processor, the situation is as in the previous section and clearly no markers need to be allocated between the two goals. But if `a` and `c` are executed on the same processor, one after the other, since they are known to be deterministic, no markers need to be allocated between them either. This is based on the fact that if `a`, `b`, `c` are known to be deterministic and independent `a & b & c` is equivalent to `a & c & b`, and to any other permutation, modulo side effects.[5] The advantage is clearly that the input marker of `a` can be simply shared by `c`.

The optimization can also be applied in cases where whole collections of related deterministic goals are created in loops, as in

```
p1:- a & b & p.
p2.
```

We assume that `a, b, p` are known to be deterministic. An execution of `p` would generate goals of the form

```
a & b & a' & b' & a'' & b'' ...  & p2
```

Since all these goals are independent and deterministic, no intermediate marker is needed whenever they stack one over the other in a given processor, i.e., only one marker

[5] Note that side effects would have been taken into account beforehand by the parallelizer by imposing a synchronization among them — which, in the worst case, can lead to the sequentialization of the goals.

would be needed per processor (assuming there are no other parallel conjunctions). Note that when `p` is to be backtracked from outside, all the goals `a, b, a', b'...` have to be backtracked over. However, the order in which this is done is not important. Thus, every segment formed by consecutively stacked goals can be backtracked (only untrailing is really required) in one step by simply unwinding down to the sole input marker. This saves time and space in forward execution, since fewer markers are needed, and time in backward execution, since fewer intermediate steps and messages are needed. We will call the set of parallel goals `a, b, a', b', a'', b''...` a *backtracking family*: a set of independent parallel goals, such that all of them are backtracked over in the same backtracking step in the sequential execution. The fundamental characteristic of the members of a backtracking family is that they have a "common choice point" which they backtrack to in case of failure.

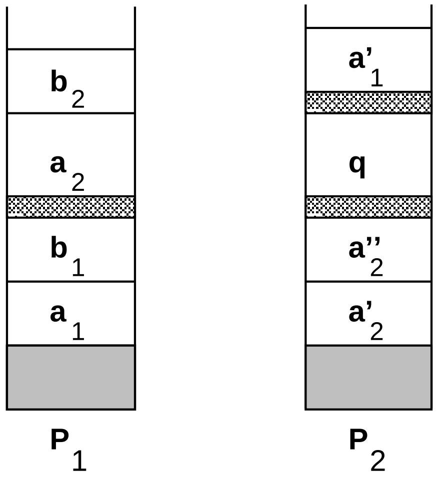


Figure 5.2: Backtracking families

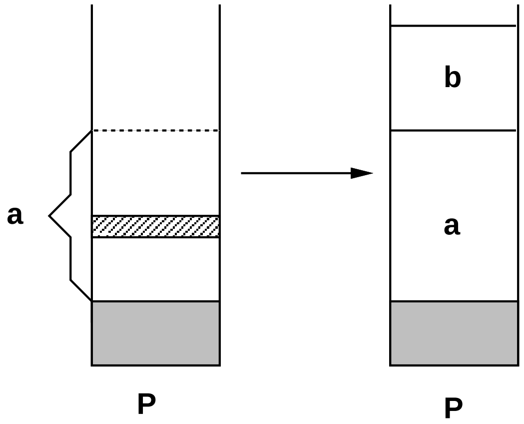


Figure 5.3: Eliminated choice points

As mentioned before, unlike those proposed previously, this technique requires knowledge regarding goal determinism beforehand. In order to illustrate this, consider the situation in which, for `a & b & c`, `a` is executed on processor P_1, and `b` in P_2. P_1 (which is deterministic) finishes first with `a` and picks up the goal corresponding to `c` (also deterministic). If `b` is deterministic, then there is no need for any marker between `a` and `c` (or saving the trail segment in the slot) since no intermediate backtracking is possible. This may be determined once `b` finishes, but then P_1 would have to wait for P_2, which is undesirable. On the other hand, if P_2 had finished before, then it could have stolen `b`, and the determinate processor optimization could have been applied.

Each backtracking family is given a unique identifier (e.g., the address of the "common choice point" that they would backtrack to). This identifier is also associated with each goal belonging to the family and stored in the input marker when the first goal of a family is picked up by a processor. When a goal is picked up by a processor, if it has a family identifier attached and it is the same as that of the current input marker in the processor, no new input marker needs to be allocated. It is clear that this has an implication on scheduling in the sense that picking up goals belonging to the same backtracking family as the last goal executed in a given processor is always preferable.

Markers are still necessary in principle between deterministic and non–deterministic goals, and between goals that do not belong to the same backtracking family. If we have $\mathtt{p}_1$ `&` `q` `&` $\mathtt{p}_2$, where `p` is defined as above, and `q` generates non–deterministic computations, then the goals generated by $\mathtt{p}_1$ can be stacked one over the other without markers. However, markers are needed to separate goals generated by $\mathtt{p}_1$ and `q`, $\mathtt{p}_2$ and `q`, and $\mathtt{p}_1$ and $\mathtt{p}_2$ (this is illustrated in Figure 5.2, where segments marked with $'$, $''$... correspond to different activations of the same goal, and goals marked with a subindex are offsprings of the corresponding initial call, $\mathtt{p}_1$ or $\mathtt{p}_2$).

However, note that the optimization is not necessarily restricted to deterministic goals, as might be implied by the discussion above. In fact, the fundamental characteristic of the goals of a backtracking family is that they have a "common choice point" that they backtrack to in case of failure of one of such goals. Thus, if a goal is deterministic in the end (i.e., it produces only one solution) it can also benefit from the proposed technique, even if it does create choice points and backtrack internally along the way, provided that it can be determined that such choice points will not provide additional solutions or that they will be discarded upon termination (for example, by executing a "cut"). In summary, goals which can be determined to have a single solution, independently of whether they create choice points during their execution and perform backtracking internally, are also eligible for forming a backtracking family with other such goals or with deterministic goals. Simple examples are `(a, !) & (b, !)`, or even `(a & b), !`, where `a` and `b` may have non–determinism inside, but are made single solution by the presence of the cut (see Figure 5.3). As another, more elaborate example, consider sorting a list of complex items (i.e., not simple numbers) with quick-sort, where the tests performed in the partitioning predicate could be arbitrarily complex, leaving intermediate choice points and backtracking internally, but finally yielding only one solution. We believe it is possible to detect this "single solution" status in many cases through existing compile-time analysis techniques, even if this cannot be determined locally, as in the simple examples above.

In order to implement the proposed optimization the determinism information is passed to the low-level compiler through source program annotations. In the same way as we have assumed for the “&” annotations, these determinacy annotations can be provided by the user or generated by an automatic analyzer. Note that, while the annotation is static, its effect has a dynamic nature in general in the sense that, for a given program, the actual performance increase may differ from execution to execution due to different schedulings, which would result in different relative stackings of members of different families, and thus different actual numbers of markers allocated.

Regarding the benefits obtainable from the optimization proposed, it can clearly provide considerable savings in memory consumption, and, as a side effect of this, time savings due to the smaller number of markers which have to be initialized. In an ideal situation, in which all goals picked up by each processor belong to the same backtracking family only one marker would be allocated per processor (even while choice points internal to a parallel goal are allocated, used, and eventually discarded before the parallel goal finishes). Furthermore, and as mentioned before, backtracking is potentially also greatly sped up.

Clearly, the practical advantages which can be obtained automatically with the backtracking families optimization strongly depend on the quality of the compile-time analysis performed or, if done manually, of the annotations provided by the user. The general issue of static analysis of determinism is beyond the scope of this paper. However, the potential of the optimization can still be assessed by making reasonable assumptions regarding the information that could be obtained based on the current state of the art in global analysis, and annotating the programs to encode this information. We have done this for a number of benchmarks and the results are shown in Table 5.2, which shows the number of markers allocated (without and with the optimization) when executing on 10 processors.

As expected, the number of markers in the optimized version actually differs much from run to run – a range over a large number of runs is given in this case. It can be observed that, as expected, the reduction in the number of markers allocated is quite significant, and larger than with the dynamic methods studied previously (which have the obvious advantage on the other hand of not requiring analysis). The results are graphically compared in figure 5.4.

As mentioned before, the advantage comes either from the knowledge that parallel calls that do create choice-points (and thus are not eligible for the shallow backtracking optimization dynamically) are in fact deterministic, or from the knowledge that goals that are not deterministic and are picked up by a processor out of contiguous order (and thus are not eligible for the determinate processor optimization) are in the same backtracking family. For example, it is quite simple to determine by global analysis (using

Executed goal	**Unoptimized**	**Optimized**
deriv(1)	261	[9]
deriv(2)	2109	[11]
deriv(3)	16893	[11]
deriv(4)	135165	[11]
boyer(0)	24	[3 – 8]
boyer(1)	747	[111 – 153]
boyer(2)	7290	[543 – 745]
boyer(3)	282168	[4770 – 6500]
quick_sort(50)	150	[13 – 15]
quick_sort(100)	300	[14 – 16]
quick_sort(150)	450	[15 – 17]
quick_sort(200)	600	[15 – 16]
poccur(1)	30	[9 – 10]
poccur(2)	60	[11 – 12]
poccur(3)	90	[11 – 13]
poccur(4)	120	[12 – 13]
poccur(5)	150	[12 – 13]
takeuchi(13)	1412	[19 – 23]
takeuchi(14)	4744	[21 – 29]
takeuchi(15)	10736	[21 – 30]
takeuchi(16)	21236	[23 – 28]

Table 5.2: Backtracking families optimization (memory consumption, 10 proc.)

the same information that the parallelizing compiler uses to parallelize the benchmark) that the matrix benchmark is completely deterministic. This is the case also with most of the other examples in Table 5.2. However, the issue of determining precisely the exact extent to which this optimization is applicable in large programs using state of the art analysis technology remains a topic for future work.

5.4 Low Level Optimizations and Data-Parallelism

Low level optimizations (i.e., those close to the abstract machine implementation), as those proposed and evaluated in Chapter 5 can be applied to many common cases, the

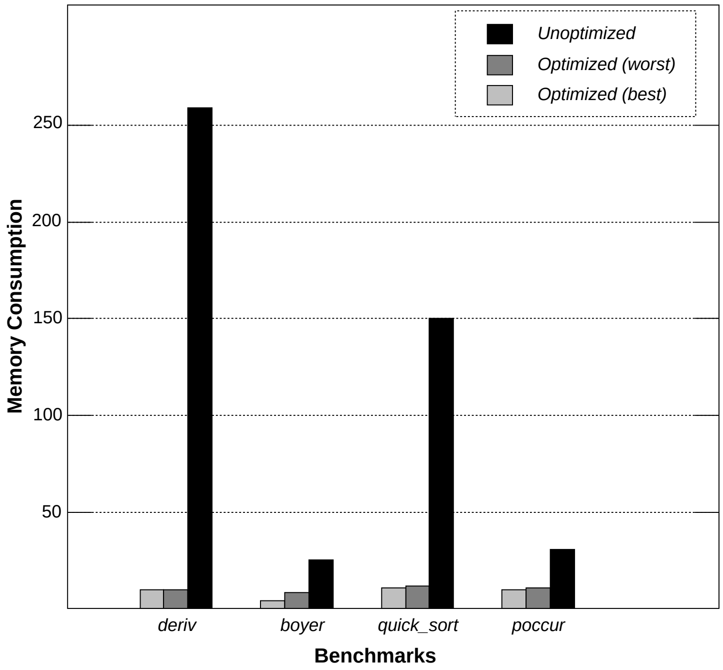


Figure 5.4: Improvements using Backtracking Families

more frequent being possibly those implementing simple loops. These are quite often deterministic (or, at least, they yield a single solution) needed to apply the optimizations proposed in that chapter.

Interestingly, these loops are also the simplest way to express computations with data-parallelism in a logic language (Chapter 4). Let us remember that one of the characteristics of data parallel computations is the determinism. There is, therefore, a clear possibility of positive interaction between data parallelism and the proposed low level optimizations: LPCO and backtracking families are applicable to the typical data-parallelism cases.

Additionally, the proposed data-parallelism transformations aimed at speeding up task creation also respect these conditions. The transformed programs in Sections 4.2.3 to 4.3 are, as the original programs, completely deterministic, so that backtracking families and LPCO can be exploited. In fact, a great deal of the test programs in Table 5.2 (in particular, *deriv*, *boyer*, *quick_sort*, and *takeuchi*) have a *divide and conquer* structure similar to what would be obtained after a binarization. Therefore, a performance improvement similar to that obtained in this table is to be expected.

A faster task startup for data parallel programs can be obtained by introducing lower-level mechanism for the specific compilation of data parallel programs. Without copying a data-parallel system, which includes these apparatus as basic pieces, it is possible to use the blocks already present in many systems featuring conjunctive parallelism in order to simulate these mechanisms. In particular, a proposal in order to modify the structure of the goal stack of a RAP system is presented in [HC96]; it is based on specifying the fast creation of several goals, and on assigning a part of a data structure to each of them.

Chapter 6

Realistic Simulation of Paralelism in Logic Programs

Chapter Summary

We present a technique to estimate accurate speedups for parallel logic programs with relative independence from characteristics of a given implementation or underlying parallel hardware. The proposed technique is based on gathering accurate data describing one execution at run–time, which is fed to a simulator. Alternative schedulings are then simulated and estimates computed for the the corresponding speedups. Such speedups can be used to compare different parallelizations of a program or to evaluate the performance of parallel systems. A tool implementing the aforementioned techniques is presented, and its predictions are compared to the performance of real systems, showing good correlation.

6.1 Introduction

In recent years a number of parallel implementations of logic programming languages, and, in particular, of Prolog, have been proposed (some examples are [HG91, AK90b, SCWY91a, She92a, Lus90]). Relatively extensive studies have been performed regarding the performance of these systems. However, these studies generally report only the absolute data obtained in the experiments, including at most a comparison with other actual systems implementing the same paradigm. This is understandable and appropriate in that usually what these studies try to asses is the effectiveness of a given implementation against state–of–the–art sequential Prolog implementations or against similar parallel systems.

In this paper, and in line with [SH91], we try to find techniques to answer a different question: given a (parallel) execution paradigm, what is the maximum benefit that can be obtained from executing a program in parallel in a system designed according to that paradigm? (we will refer to this as "maximum parallelism"). What are the resources (for example, processors) needed to exploit all parallelism available in a program? How much parallelism can be ideally exploited for a given set of resources (e.g. a fixed number of processors)? (we will refer to this as "ideal parallelism"). The answers to these questions can be very useful in order to evaluate actual implementations, or even parts of them, such as, for example, parallelizing compilers. However, such answers cannot be obtained from an actual implementation, either because of limitations of the implementation itself or because of limitations of the underlying machinery, such as the number of processors or the available memory. It appears that any approach for obtaining such answers has to resort to a greater or lesser extent to simulations.

There has been some previous work in the area of ideal parallel performance determination through simulation in logic programs, in particular the work of Shen [SH91] and Sehr [SK92]. These approaches are similar in spirit and objective to ours, but differ in the approach (and the results).

In [SH91] a method is proposed for the evaluation of potential parallelism. The program is executed by a high–level meta-interpreter/simulator which computes ideal speedups for independent and–parallelism, or–parallelism, and combinations thereof (see [Con83] and Section 6.3 for a description of different types of parallelism in logic programs). Such speedups can be obtained for different numbers of processors.

This work is interesting, firstly in that it proposed the idea of obtaining ideal performance data through simulations in order to be able to evaluate the performance of actual systems by contrasting them with this ideal and, second, because it provides ideal speedup data for a good number of programs. However, the simulator proposed does suffer from some drawbacks. The first one is that all calculations are performed using as time unit a resolution step – i.e. all resolution steps are approximated as taking the same amount of time. This makes the simulation either conservative or optimistic in programs with (respectively) small or large head unifications. To somewhat compensate for this, and to simulate actual overheads in the machine, extra time can be added at the start and end of each task. The second drawback is that the meta-interpretive method used for running the programs limits the size of the executions which can be studied due to the time and memory consumption implied.

In [SK92] a different approach was used, in order to overcome the limitations of the method presented above. The Prolog program is instrumented to count the number of WAM [War83, AK91] instructions executed at each point, assuming a constant cost for each WAM instruction. Only "maximal" speedup is provided. Or–parallel execution is simulated by detecting the critical (longest) path and comparing the length of this path with the sequential execution length. Independent and-parallel execution is handled in a similar way by explicitly taking care of the dependencies in the program. Although this method can be more accurate than that of [SH91] it also has some drawbacks. One is the fact mentioned above that only maximal speedups are computed, although this could presumably be solved with a back-end implementing scheduling algorithms such as the ones that we will present. Another is that the type of instrumentation performed on the code does not allow taking control instructions into account. Also, a good knowledge of the particular compiler being used is needed in order to mimic its encoding of clauses. Furthermore, many WAM instructions take different amounts of time depending on the actual variable bindings appearing at run-time, and this would be costly and complicated to take into account. Finally, the problem of being able to simulate large problems is only solved in part by this approach, since running the transformed programs involves non-trivial overheads over the original ones.

The approach that we propose tries to overcome the precission and execution size limitations of previous approaches by using precise timing information. Also, it allows gathering information for much larger executions. We do that by placing the splitting point between actual execution and simulation at a different location: sequential tasks are not simulated or transformed but executed directly in real systems.

Although the techniques we present have been designed within the area of parallel logic programming, we believe that the core idea can be applied to any execution paradigm, and that the techniques (and tools) developed can be applied directly to those paradigms conforming to the initial assumptions.

The paper is structured as follows: Section 6.2 sketches our objectives. Section 6.3 describes more in depth our approach and the techniques used in its implementation. Section 6.4 relates the traces obtained at run–time with the graphs used to simulate alternative schedulings. Sections 6.5 and 6.6, respectively, show how the maximum and ideal parallelism are calculated. In Section 6.7 an overview of *IDRA*, the actual tool, is given. Section 6.8 contains examples of simulations made using *IDRA* and comparisons of actual implementations with the results of the simulation.

6.2 Objectives

Our objective is to perform speedup analysis of executions of parallel logic programs, in a relatively independent way from the characteristics (such as number of processors, absolute speed, etc.) of the platform in which they have been executed. Given a (parallel) program and a number (which may be unbound) of processors, different schedulings can (and do) greatly affect the total execution time.[1]

Among the information we can extract from alternative schedulings, the following may be of interest:

- Maximum parallelism: this corresponds to the parallelism obtained with an unbound number of processors, assuming no scheduling overheads.
- Ideal parallelism: this corresponds to the speedup ideally attainable with a fixed number of processors. The tasks–processors mapping here decides the actual speedups attained. Optimal scheduling algorithms and currently implemented algorithms are clear candidates to be studied.

Maximum parallelism is useful in order to determine the absolute maximum performance of a program, i.e., the minimum time in which it could have been executed while respecting the dependencies among tasks. This is used, for example, for comparing different parallelizations/sequentializations of a given program (e.g., if different domains or annotators for parallelism are being evaluated, see for example [BdlBH94a, BdlBH94b]) or different parallel algorithms proposed for a given problem (e.g. [DJ94]). In the simulation we know that the speedup obtained has not been limited by the machine itself (e.g., number of processors, bus contention, etc.)

Ideal parallelism can be used to test the absolute performance of a given scheduling algorithm in a fixed number of processors, by comparing the speedup obtained in the machine with the maximum speedup attainable using that number of processors. The efficiency of an implementation can also be studied by testing the actual speedups against those predicted by the simulator using the same scheduling algorithm as the implementation. Also, how the performance of a program evolves for a number of processors as large as desired can be studied; this gives interesting information about the potential parallelism in a program.

We want our simulation to be useful for medium size applications, and the results to be as accurate as possible. That is why the simulation takes place at the scheduling level, the sequential task timing being (preferably) obtained using real executions.

[1] Of course, faster processors will affect the absolute execution time as well, but since ideally this speed scales to the whole execution, the speedup obtained with respect to sequential executions should not change. This, in practice, is not true, since, for example, bus bandwidth limits the attainable speedup in memory–intensive applications. This is, precisely, one example of the limitations we want to overcome.

6.3 Parallelism and Trace Files

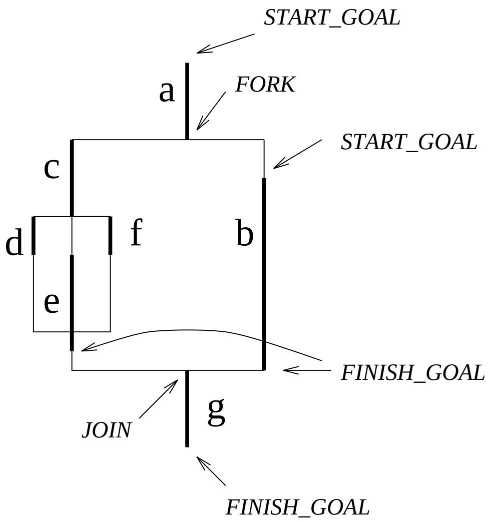
To simulate an alternative scheduling of a parallel execution we need a certain description of that execution. This description must contain, at least, the relationships and dependencies which hold among the tasks (used to simulate new correct schedulings, i.e., executions where the precedence relationships are met), and the length (in time) of each task. Such a description can be produced by executing programs in actual implementations (not necessarily parallel ones: only the description of the concurrency in the execution and each task's length must appear, the parallelism among tasks being introduced by means of the simulation) augmented to generate execution logs, or even using other high–level simulators able to produce information about dependencies in the program and an estimation of the (relative) cost of executing each sequential task. This considerably widens the applicability of the developed tool because it allows studying the (expected) performance of parallel programs and scheduling algorithms without the need of an actual parallel machine or in non–realistic conditions (for example, unbound number of processors).

The format and components of the traces, as well as the way they are generated and stored, are the same as those used in *VisAndOr* (Section 3.8). Events (Table 3.2) reflect observable in the execution, and each even carries information enough as to establish dependencies with other events in the same execution. In our case the timing data has been obtained from a modified Prolog implementation which ensures that the simulation is realistic. The simulation concerns generating different schedulings of the sequential tasks which respect their precedences.

It should be noted that, in parallel dialects of Prolog, collecting traces is easy from the user point of view. The structure of the Prolog language and its implicit control helps to automatically identify the "interesting places" (for example, where a sequential task starts or finishes) in the execution. The parallel execution models which we will deal with in this paper stem naturally from the view of logic programming as a process–oriented computation. The two main types of parallelism available in a logic program are And–parallelism and Or–parallelism, which have already been reviewed in Chapter 2.

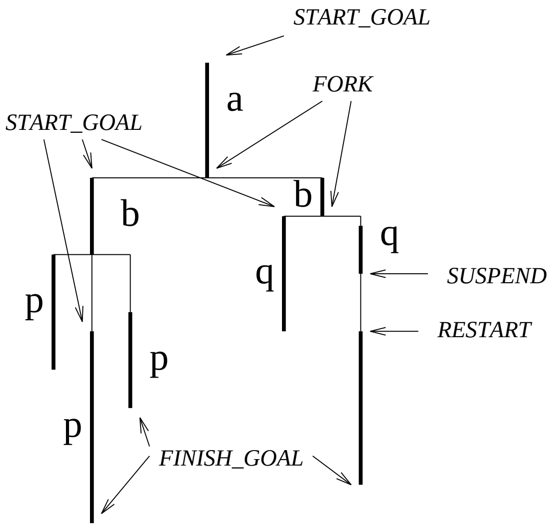
6.4 From Traces to Graphs

From a practical point of view, the format of the traces may depend on the system that created them: traces may have information that is not necessary, or be structured in




```
main:- a, c & b, g.
c:- d & e & f.
```

Figure 6.1: And–parallel execution




```
main:- a, b.
b:- p.
b:- q.

q:-  ...
q:-  ...

p:-  ...
p:-  ...
p:-  ...
```

Figure 6.2: Or–parallel execution

an undesirable way, perhaps because they may serve to other purposes as well.[2] On the other hand, scheduling algorithms are usually formulated in terms of the well–known *job graphs* (see, e.g., [MC69, LL74, Hu61, HB88]). However, in job graphs only tasks and relationships are reflected: scheduling delays do not appear—or are assumed to be a part of the tasks themselves. To be able to change the delays introduced by the scheduling algorithms, and to somewhat separate the traces from the internal structures, we will

[2]This is the case for the actual parallel systems that we study—see Section 6.7—where traces originally designed for visualization are used by our simulation.

use *execution graphs* as an intermediate object that abstracts the trace containing only the information needed to simulate new schedulings.

6.4.1 The Execution Graph

An execution graph translates the idea of events and their dependencies into a mathematical object. An execution graph is a directed acyclic weighted graph $G(X, U, T)$ where:

$X = \{x_0, x_1, \ldots, x_{n-1}\}$ is a set of nodes,

$U = \{u_{i,j}, 0 \leq i < j < n\}$ is the set of edges connecting node x_i to node x_j, and

$T = \{t_{i,j}, 0 \leq i < j < n\}$ is the set of weights labeling each $u_{i,j}$.

In the execution graph each node corresponds to an event, and has associated a type (the same of the event—see Table 3.2) and the point in time in which the corresponding event has occured. Each edge reflects a dependency between events, and its associated weight represents the time elapsed between them. We distinguish two types of edges: those which represent the sequential execution of a task and those which represent delays introduced by scheduling. The edges fall, thus, in one of the following two categories:

Scheduling Edges: FORK to START_GOAL, FINISH_GOAL to JOIN.

Execution Edges: START_GOAL to FINISH_GOAL, START_GOAL to FORK, JOIN to FINISH_GOAL, JOIN to FORK.

Figures 6.3 and 6.4 show, respectively, the structure of the execution graphs corresponding to the traces depicted in Figures 6.1 and 6.2 (the weights have been omitted for simplicity). The execution graphs is a formal, intermediate representation for event traces. This representation is transformed into a job graph, wich is in turn used to simulate the schedulings.

6.4.2 The Job Graph

A job graph $G(X, U)$ consists of a set of nodes $X = \{x_0, \ldots, x_{n-1}\}$ and a set of edges $U = \{u_{i,j}, 0 \leq i < j < n\}$, where each $u_{i,j}$ represents an edge from node x_i to x_j. The graph contains a node for each sequential task in the execution and an edge for each dependency between tasks. Each node x_i has information related to the task it represents, such as its length $l(x_i)$ and its starting time $t(x_i)$. There is a partial ordering

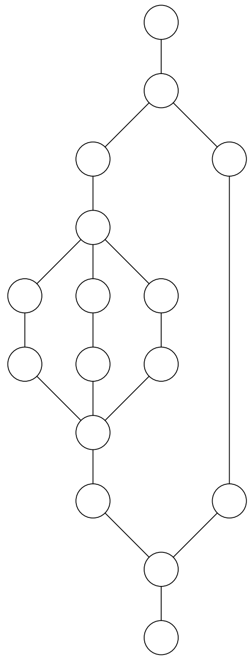


Figure 6.3: Execution graph, and–parallelism

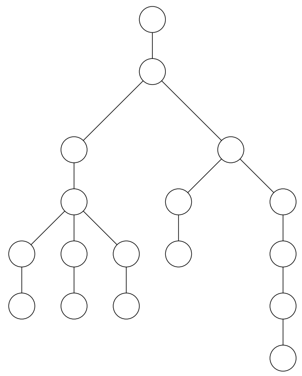


Figure 6.4: Execution graph, or–parallelism

$\prec$ among the tasks in X given by the dependencies present in the execution. We will say that $x_i \prec x_j$ iff $u_{i,j} \in U$. Figures 6.5 and 6.6 show job graphs for the and– and or–parallel examples we have been using throughout the paper.

Job graphs are obtained from execution graphs by eliminating the scheduling times (represented by scheduling edges) and transforming the execution edges (which represent actual sequential tasks) into nodes. The dependencies in the job graph and the length of each task are inherited from the execution graph. This transformation can, of course, be parameterized to take into account actual or minimal scheduling delays,

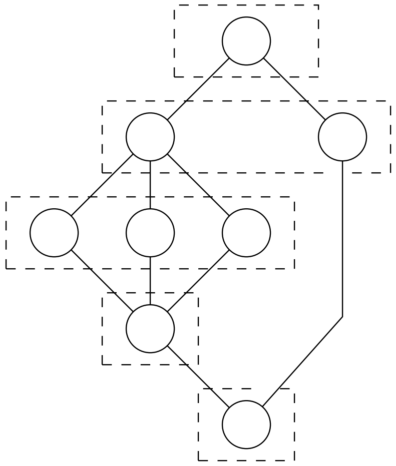


Figure 6.5: Job graph for and–parallelism

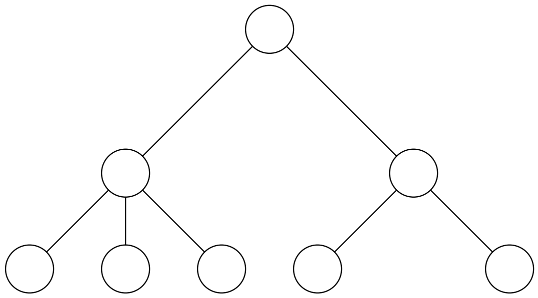


Figure 6.6: Job graph for or–parallelism

incrementing the usefulness of the tool.

6.4.3 Scheduling in Job Graphs

A scheduling for a given execution $G(X, U)$ can be formally viewed as a function $\sigma : X \to \mathcal{Z}^+$ that assigns a starting time to each task, the task's length remaining unchanged. In order for σ to represent a correct scheduling, no task can start before all its predecessors have finished:

$$\forall x_i, x_j \in X : x_i \prec x_j \to \sigma(x_i) + l(x_i) \leq \sigma(x_j) \tag{6.1}$$

A scheduling σ that minimizes the time spent in the execution has to meet the following condition:

$$\begin{aligned}&\text{Let } L_{\sigma'} = \max_{x \in X}(\sigma'(x) + l(x)) \text{ for a } \sigma' \in \{X \to \mathcal{Z}^+\}.\\ &\text{Then } \sigma \text{ is such that } L_\sigma = \min_{\sigma' \in \{X \to \mathcal{Z}^+\}}(L_{\sigma'})\end{aligned} \tag{6.2}$$

Scheduling algorithms can be classified depending on whether they are deterministic (used when all data pertaining the execution is available [MC69, LL74, Hu61]) or non deterministic (in which random variables with known characteristic functions are used to model non available data [HB88]). Since we are doing "post–mortem" scheduling simulations, our case is the former.

6.5 Maximum Parallelism

As mentioned in Section 6.2, maximum parallelism assumes a null scheduling time and an infinite number of processors, so that newly generated tasks can be started without any delay at all. A scheduling with these conditions can be modeled as a function σ, as described in Section 6.4.3 and which meets conditions 6.1 and 6.2.

Two interesting results we can obtain from a simulation with these characteristics are the maximum speedup attainable and the minimum number of processors needed to achieve it. Obtaining both these numbers is an NP–complete problem [GJ79]; however, the exact maximum speedup and an upper bound on the number of processors is easy to obtain. This is still useful, because it gives an estimation of the best performance that can be expected from the program(s) under study. It can serve to compare alternative parallelizations of a program, without the possible biases and limitations that actual executions impose, but still retaining the accuracy in the timing of the tasks.

We can recalculate the starting time $t(x)$ assigned to each node $x \in X$, starting at 0 for the first node in the execution, so that the starting time in each task corresponds to the maximum of the ending time of its predecessors. Then, assuming that x_{n-1} is the node corresponding to the last task in the execution, the minimum time that the execution can take is $t(x_{n-1}) + l(x_{n-1})$. From this, speedups with respect to sequential executions are straightforward to obtain, the sequential execution time being the sum of the lengths of all the tasks.

The maximum number of tasks simultaneously active is an upper bound on the minimum number of processors needed to achieve this execution time. Let $N(t)$ be defined as

$$N(t) = |\{x \in X | t(x) \leq t \leq t(x) + l(x)\}| \tag{6.3}$$

i.e., $N(T)$ is the number of tasks active at time t. The minimum number p of processors needed to execute without delays is

$$p = \max_{0 \leq t < t(x_{n-1}) + l(x_{n-1})} N(t)$$

assuming again that the program execution starts at time 0.

Note that high speedups do not necessarily mean that the program is a good candidate for parallel execution: this depends, of course, on the number of processors at which this maximum parallelism is achieved. We will see examples illustrating this in Section 6.8.2.

6.6 Ideal Parallelism

By ideal parallelism we refer to the situation in which, for a given number m of processors, a perfect scheduling has been performed, in the sense that the minimum execution time possible (with that number of processors) was achieved. A scheduling algorithm that performs ideal parallelism can be modeled by a mapping σ as defined in 6.4.3, to which the following restriction has been added:

$$\forall t, 0 \leq t < t(x_{n-1}) + l(x_{n-1}), N(t) \leq m \tag{6.4}$$

where $N(t)$ is as defined in equation 6.3, i.e., the number of tasks simultaneously active is less than or equal to m.

Such σ gives the optimum starting time for each task. From it, a processor–task mapping is straightforward, since it is required that no more than m tasks be active at a time. When a task is finished, the processor that executed it can be assigned to the task with the nearest starting time.[3]

It would be interesting to find out the speedups achievable using a perfect scheduling. Unfortunately, obtaining an optimal task/processor allocation is, in general, an NP–complete problem [GJ79]. Since we want to deal with sizeable, non trivial, programs, this option is too computationally expensive to be used. Instead, we will employ non optimal scheduling algorithms which give an *adequate* (able to compute a reasonable answer for a typical input), but not *appropriate* (every processor is attached to a sequential task until this task is finished) scheduling.

From a high level point of view, the ideal parallelism simulation takes a description of the execution, a scheduling algorithm A, and a number of processors N, and returns

[3]Under the implicit assumption that any processor is able to execute any task.

the maximum speedup attainable in the form of a function $t : X \rightarrow \mathcal{Z}^+$ that reflects the calculated starting time for each task.

The algorithm we implemented to find out quasi–optimal schedulings is the so–called *subsets* [HB88] algorithm, which in fact gives optimal results under certain conditions (that are however not always met in our more general case).

Testing the quality of an existing scheduler against an idealized one is also interesting, because that comparison would give an idea of how good is the implementation of the scheduling algorithm. Following that idea, we also implemented a version of the scheduling scheme found in the *&–Prolog* system [HG91, Her87]. We expect the comparison of the actual *&–Prolog* system speedups and the results obtained from *IDRA* to serve as an assessment of the accuracy of our technique, whereas the comparison among a (quasi–)optimal scheduling and a real one would serve to estimate the performance of the actual system.

The variation of the inherent parallelism with the problem size is also a topic of interest. Frequently one wants more performance not only to solve existing problems faster, but also to be able to tackle larger problems in a reasonable amount of time. In simple problems the number of parallel tasks and the expected attainable speedups can be calculated, but in non–trivial examples it may not be so easy to estimate that. Problems in which available parallelism does not increase with the size of the problem would not benefit from a larger machine. In Section 6.8 examples illustrating this are given.

In the next two sections we will describe the two scheduling algorithms currently implemented in the simulation tool.

6.6.1 The *Subsets* Scheduling Algorithm

The **subsets** [HB88] algorithm avoids performing a global scheduling by splitting the nodes (tasks) in the job graph into disjoint subsets (those inside dashed rectangles in Figure 6.5). The nodes in each subset represent tasks that are independent among them, and so they are candidates for parallel execution.

Each processor $j, 0 \leq j < p$, is modeled as a number T_j which represents the moment from which it is free to execute new work. The set $P = \{T_0, \ldots, T_{p-1}\}$ contains the availability times of the processors in the system. At any given time, no task can be scheduled before $\min_{T \in P}(T)$.

The initial subset is a singleton containing only the first task: $S_0 = \{x_0\}$, and for each subset S_i, S_{i+1} is the set of nodes which can start once all the nodes in S_i have finished.

If all the tasks in subset S_{i+1} started after the last task in S_i finish, the subsets could have been scheduled independently. Since a given task in S_{i+1} may depend only on some of the tasks in S_i, the starting time of each task in S_{i+1} is set to the time in which all their predecessor tasks in S_i have finished. In each subset $S_i = \{t_1, \ldots\}$, the scheduling algorithm assigns one task t_j to one processor from P. For each subset $S \neq S_0$, the algorithm performs as follows:

For each task $t_j \in S$ do:

Step 1 Let $Time_j = \max_{x \in X, x \prec t_j}(t(x))$. This is the earliest time in which t_j can start.

Step 2 If there is any processor p such that $T_p \leq Time_j$, assign processor p to task t_j, and set $T_p = T_p + l(t_j)$ and $t(t_j) = Time_j$.

Step 3 Otherwise, find $T_q = \min_{T \in P}(T)$. Assign task t_j to processor q and set $T_q = T_q + l(t_j)$.

Tasks are assigned to free processors. If no free processor exists at a given moment, the first processor to become idle is chosen. The need to make a choice in the non–deterministic Step 2 is one of the sources of the non optimality of the algorithm. In Step 3, T_q is chosen using a heuristic that tries to increase the occupation time of the processors.

6.6.2 The *Andp* Scheduling Algorithm

The **andp** scheduling algorithm [Her87] mimics the behavior of one of the *&–Prolog* schedulers. For each processor, *&–Prolog* has the notion of *local* and *non local* work: local work is the work generated by a given processor, and it is preferably assigned to it. To keep track of the local work, each processor is modeled as a couple $\langle T, L \rangle$ where T is as before, and L is the list of tasks generated by the processor. Roughly speaking, the scheduling algorithm tries first to execute tasks locally; if this is not possible, a task is stolen from another processor's list.

The **andp** scheduling algorithm can be split into two different parts: the first one takes care of obtaining work available in the system, and the second one generates new work and stores it in the processor's local stack.

Processor 0 is selected as having the initial task; thus at the beginning $L_0 = \{x_0\}$. The rest of the processors have empty stacks: $L_i = \emptyset, 0 < i < p$, and all of them are free: $T_i = 0, 0 \leq i < p$. The part of the scheduling algorithm that is in charge of getting work is as follows:

Step 1 If $\forall \langle T_i, L_i \rangle \in P,\ L_i = \emptyset$, finish. Otherwise select the processor p such that $T_p = \min_{\langle T_i, L_i \rangle \in P}(T_i)$

Step 2 If $L_p \neq \emptyset$ assign the first task $x \in L_p$ to processor p and go to Step 1.

Step 3 If $L_p = \emptyset$, find the processor q such that $T_q = \min_{\langle T_i, L_i \rangle \in N, L_i \neq \emptyset}(T_i)$. Assign the first task $x \in L_q$ to processor p and go to Step 2.

The generation of new work, after task x from the list of tasks L_q is assigned to processor p, is as follows:

Step 1 Set $L_q = L_q - \{x\}$.

Step 2 Set $T_p = T_p + l(x)$.

Step 3 Set $L_p = L_p \cup \{x_i \in X \text{ s.t. } x \prec x_i\}$.

6.7 Overview of the Tool

A tool, named *IDRA* (IDeal Resource Allocation), has been implemented using the ideas and algorithms shown before. The traces used by *IDRA* are the same as those used by the visualization tool *VisAndOr* [CGH93]. Thus, *IDRA* can calculate speedups for the systems *VisAndOr* can visualize (namely, the independent and–parallel system *&–Prolog* and the or–parallel systems Muse and Aurora — the deterministic dependent and–parallel system Andorra–I is not supported yet — as well as others which implement parallelism of a similar structure) using directly the trace files that *VisAndOr* accepts, without the need of any further processing.

The tool itself has been completely implemented in Prolog. Besides the computation of maximum and ideal speedups, *IDRA* can generate new trace files for ideal parallelism, which can in turn be visualized using *VisAndOr* and compared to the original one. *IDRA* can also be instructed to generate automatically speedup data for a range of processors. This data is dumped in a format suitable for a tool like xgraph to read.

The traces used with *IDRA* (and with *VisAndOr*), need not be generated by a real parallel system. This is a very interesting feature, in that it is possible to generate them with a sequential system augmented to dump information about concurrency. The only requirement is that the dependencies among tasks be properly reflected, and that the timings be accurate.

In some platforms accuracy in the timings has not been straightforward to obtain. Somo usual UNIX environments do not provide good access to the system clock: calls to standard OS routines to find out the current time either were not accurate enough for our purposes, or the time employed in such calls were a significant portion of the total execution time of the benchmark, thus leading to incorrect results (sequential tasks being traced were noticeably longer than without tracing). To obtain accurate timings we used the microsecond resolution clock available in some Sequent multiprocessors [Seq87], which is not only very precise, but also memory mapped and can thus be read in the time corresponding to one memory access, with negligible effect on performance. For platforms in which the clock has a high but predictable access time, we had to develop a technique based on subtracting the accumulated clock access time from the timings.

The overhead of gathering the traces depends ultimately on the system executing the program being traced. For the *&–Prolog*/Muse systems, it typically falls in the range 0% – 30% — usually less than 20% — of the total execution time.

The time that a simulation takes depends, of course, on the trace being inspected. It can be substantially larger than the execution itself if the program executes many small tasks, and can be shorter than executing the actual program in the opposite case: few, large tasks.

6.8 Using *IDRA*

In this section we will show examples of the use of *IDRA* on real execution traces. The traces we will use have been generated by the *&–Prolog* system for and–parallelism, and by *&–Prolog* and Muse for or–parallelism. The generation of the traces corresponding to or–parallelism needed of a slight modification of *&–Prolog* to make it issue an event each time a choice-point is created.

The reason to generate or–parallel traces using *&–Prolog* was that or–parallel schedulers (and that of Muse in particular) usually make work available to parallel execution only for some choicepoints. This, in our approach, would not allow us to find out the maximum or ideal parallelism hidden in the program, since opportunities for performing work in parallel would be lost. This is why *&–Prolog*-generated or–parallel traces achieve better speedups than the corresponding ones generated by Muse: more tasks can be scheduled for parallel execution. On the other hand, the reason why the Muse scheduler does not schedule all possible tasks for parallel execution is that the added overhead would possibly result in poorer speedups.

6.8.1 Description of the Programs

We include a brief description of the programs used to test the tool, in order to help in understanding their behavior, both in simulation and in execution. The sequential execution time and the number of tasks generated by each benchmark program are shown in Table 6.1, as an indication of the program size. The figures that appear next to some

Program	**Time (ms)**	**Number of tasks generated**	
deriv	240	2109	
occur	1750	126	
tak	610	4744	
boyer	110	747	
matrix–10	170	321	
matrix–15	550	726	
matrix–20	1270	1270	
matrix–25	2460	2047	
quicksort–400	590	1230	
quicksort–600	1070	1500	
quicksort–750	1500	1700	
bpebpf–30	220	1395	
bpesf–30	180	90	
pesf–30	200	93	
		&–Prolog or traces	Muse traces
domino	130	1002	340
queens	70	458	176
witt	5090	1878	230
lanford1	160	458	130
lanford2	2090	2047	832

Table 6.1: Some information about each benchmark program

of the benchmark names represent the size of the input data: for **matrix**, the number of rows and columns of the matrix to be multiplied; for **quicksort**, the length of the list to be sorted, and for **bpebpf**, **bpesf** and **pesf**, the number of factors in the series.

- Programs with and–parallelism

deriv performs symbolic derivation.

occur counts occurrences in lists.

tak computes the Takeuchi function.

boyer adaptation of the Boyer–Moore theorem prover.

matrix square matrix multiplications (the vector times vector multiplications are sequential tasks).

quicksort standard quicksort program, using *append/3* instead of difference lists.

bpebpf calculates the number e, using the series $e = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \cdots$ A divide–and–conquer scheme is used both for the series and for each of the factorial

Program	Speedup	Processors	Efficiency
deriv	100.97	378	0.26
occur	31.65	49	0.64
tak	44.16	315	0.14
boyer	3.49	11	0.31
matrix–10	26.86	80	0.33
matrix–15	58.70	170	0.34
matrix–20	101.91	286	0.35
matrix–25	161.68	462	0.34
quicksort–400	3.93	15	0.26
quicksort–600	4.07	17	0.23
quicksort–750	4.28	19	0.22
bpebpf–30	23.21	260	0.08
bpesf–30	10.11	31	0.32
pesf–30	2.59	25	0.10

Table 6.2: Estimated maximum and–parallelism

calculations. This causes the generation of a very large number of tasks.

bpesf is similar to above, but each factorial is computed sequentially. The number of tasks is much smaller than above.

pesf also calculates e using the same series, but here each factor is computed in parallel with the rest of the series, from left to right.

- Programs with or–parallelism:

domino calculates all the legal sequences of 7 dominoes.

queens computes all the solutions to the 5 queens problem.

witt is a conceptual clustering program.

lanford1 determines some elements needed to complete a Lanford sequence.

lanford2 similar to **lanford1**, but with different data structures.

6.8.2 Maximum Parallelism Performance

Tables 6.2 and 6.3 show the maximum speedup attainable according to the simulation, the number of processors at which this speedup is achieved, and the relative efficiency with respect to a linear speedup, i.e., efficiency = $\frac{\text{speedup}}{\text{processors}}$ for the programs mentioned above.

Program	Speedup	Processors	Efficiency
domino	32.01	59	0.54
queens	18.14	40	0.45
witt	1.12	25	0.04
lanford1	19.72	44	0.44
lanford2	114.87	475	0.24

Table 6.3: Estimated maximum or–parallelism

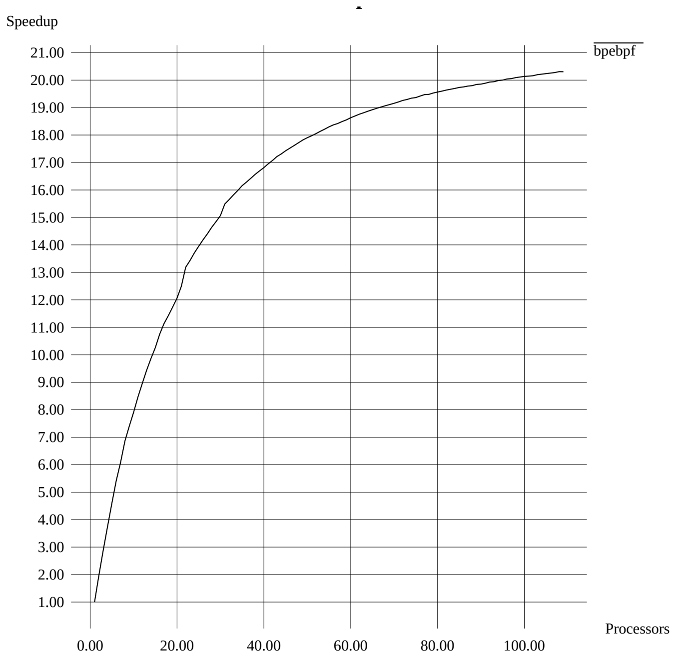


Figure 6.7: Computation of e

Programs which require a large number of processors despite the problem to be solved not being very big are usually those where tasks are small. This would suggest that a parallel system would need some sort of granularity control to execute them efficiently. This turns out not to be always the case for real executions on shared memory multiprocessors with a small number of processors,[4] as we will see in Section 6.8.3 and Table 6.4, but will certainly be an issue in larger or distributed memory machines.

In programs with a regular structure, such as **matrix**, potential speedups grow accordingly with the size of the problem, which in turn determines the number of tasks available. However, in programs where the length of the tasks is variable and the exe-

[4]In addition, *&–Prolog* concept of local work allows to speed up programs with small granularity, since stealing local tasks is much cheaper than stealing foreign tasks

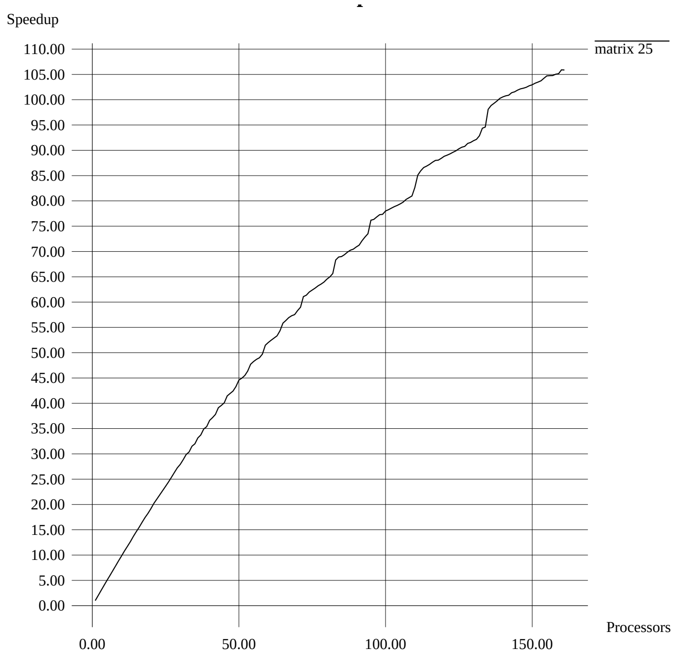


Figure 6.8: 25×25 matrix multiplication

cution structure is not homogeneous (i.e., **quicksort**), the expected maximum speedup achievable grows very slowly with the size of the problem. In the case of **quicksort**, the sequential parts caused by the partitioning and appending of the list to be sorted finally dominate the whole execution, preventing further speedups and giving an example that confirms once again Amhdal's law.

6.8.3 Ideal Parallelism Performance

For each benchmark we have determined the ideal parallelism and the actual speedups on one to nine processors (Table 6.4 and 6.5). For each benchmark, the rows marked *real* correspond to actual executions in real systems (*&–Prolog* for the and–parallel benchmarks, and Muse for the or–parallel ones). The rows marked *subsets* and *andp* correspond to simulations performed using those algorithms. There are two additonal subdivisions for each benchmark in the or–parallel case, under the column "Tracing System", which reflect in which system were gathered the traces.

The data obtained for and–parallelism with *&–Prolog* was gathered using a version of the scheduler with reduced capabilities (for example, no parallel backtracking was supported) and a very low overhead, so that the **andp** simulation and the actual execution be as close as possible. In general the results from the simulation are remarkably close to those obtained from the actual execution, which seems to imply that the simulation results are quite accurate and useful. Usually, the results with the **subsets** scheduling algorithm are slightly better, but due to its non optimality, it is surpassed sometimes by the **andp** algorithm and by *&–Prolog* itself (see, for example, the row corresponding to the quicksort–750 benchmark). With respect to the relationship between the speedups obtained by the **andp** algorithm and the actual *&–Prolog* speedups, sometimes the actual speedups are slightly better than the simulation and sometimes they are not, but in general they are quite close. This is understandable, given the heuristic nature of these algorithms.

Benchmarks that show good performance in Tables 6.2 and 6.3 have good speedups here also. But the inverse is not true: benchmarks with low performance in maximum parallelism can perform very well in actual executions (see, for example, the data for **bpebpf**). Figure 6.7 shows the simulated speedups for the benchmark **bpebpf**; Figure 6.8 shows a similar figure for **matrix** multiplication. The speedup in the first one, although showing a logarithmic behavior, is quite good for a reduced number of processors. The second one has a larger granularity and shows almost linear speedups with respect to the number of processors. When the number of processors increases beyond a limit, the expected sawtooth effect appears due to the regularity of the tasks and their more or less homogeneous distribution among the available processors.

Concerning the data for or–parallelism, Muse performs somewhat worse than the prediction given by the simulation when *&–Prolog or* traces are used. This is not surprising, since Muse has an overhead associated with task switching (due to copying) that is not reflected in the traces. Moreover, the traces correspond to the case in which all *or* branches are available for parallel execution, whereas the traces generated by Muse only contain the branches that the Muse scheduler considered worthwhile for parallel execution. Thus, in the case of the *or* traces generated by *&–Prolog*, more (and smaller) parallel tasks (and potential parallelism) exist – thus the higher speedups predicted by the tool, which largely surpass those obtained from real executions. In the case of simulations using Muse traces, the predictions are more accurate (but then, they do not reflect the parallelism available in the benchmark, but rather that exploited by Muse). In general, the results show the simulation to be highly accurate and reliable. In fact, the system has been used successfully in several studies of parallelizing transformations [DJ94] and parallelizing compilers [BdlBH94b].

6.9 Conclusions and Future Work

We have reported on a technique and a tool to compute ideal speedups using simulations which uses as input data information about executions gathered using real systems. We have applied it to or– and independent and–parallel benchmarks, and compared the results with those from actual executions. The results show that the simulation is quite reliable and corresponds well with the results obtained from actual systems, in particular with those obtained from the *&–Prolog* system. This corresponds with expectations, since the particular version of the *&–Prolog* systems used has very little overhead associated with parallel execution.

The technique can be extended for other classes of systems and execution models (also beyond logic programming), provided that the data which models the executions can be gathered with enough accuracy.

We plan to modify the simulator in order to support other execution paradigms, such as Andorra–I [SCWY91a], ACE [GHPSC94], AKL [JH91], IDIOM [GSCYH91], etc. and to study other scheduling algorithms. Finally, we believe the same approach can be used to study issues other than ideal speedup, such as memory consumption, copying overhead, etc.

Program	Scheduling Algorithm	Processors								
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
deriv	subsets	1.00	1.99	2.99	3.97	4.95	5.93	6.90	7.86	8.82
	andp	1.00	1.99	2.97	3.94	4.86	5.77	6.79	7.56	8.40
	real	1.00	2.00	3.00	4.00	4.80	4.80	6.00	8.00	8.00
occur	subsets	1.00	1.99	2.97	3.97	4.49	5.14	5.96	7.10	8.73
	andp	1.00	1.99	2.55	3.28	3.97	4.45	5.12	5.92	7.08
	real	1.00	1.96	2.96	3.97	4.48	5.83	5.83	7.00	8.75
tak	subsets	1.00	1.99	2.97	3.93	4.86	5.77	6.65	7.51	8.33
	andp	1.00	1.97	2.95	3.91	4.85	5.76	6.57	7.54	8.30
	real	1.00	1.90	2.65	3.58	4.35	5.08	5.54	6.09	6.77
boyer	subsets	1.00	1.78	2.34	2.65	2.84	2.94	3.05	3.09	3.13
	andp	1.00	1.79	2.37	2.76	3.02	3.15	3.25	3.30	3.31
	real	1.00	1.57	1.83	2.20	2.20	2.20	2.20	2.20	2.20
matrix–10	subsets	1.00	1.98	2.91	3.86	4.74	5.57	6.41	7.26	8.02
	andp	1.00	1.97	2.70	3.59	4.59	5.21	6.09	6.86	7.54
	real	1.00	1.88	2.83	3.39	4.25	5.66	5.66	6.80	8.50
matrix–15	subsets	1.00	1.99	2.96	3.94	4.91	5.84	6.76	7.71	8.62
	andp	1.00	1.97	2.85	3.51	4.40	5.36	6.37	7.15	7.84
	real	1.00	1.96	2.89	3.92	4.58	5.50	6.87	7.85	7.85
matrix–20	subsets	1.00	1.99	2.98	3.97	4.94	5.92	6.88	7.85	8.80
	andp	1.00	1.99	2.78	3.56	4.36	5.23	6.07	6.95	8.01
	real	1.00	1.95	2.95	3.84	4.88	5.77	6.68	7.47	8.46
matrix–25	subsets	1.00	1.99	2.98	3.98	4.97	5.94	6.92	7.91	8.88
	andp	1.00	1.97	2.73	3.51	4.44	5.54	6.41	7.34	7.98
	real	1.00	1.98	2.96	3.96	4.91	5.85	6.83	7.93	8.78
quicksort–400	subsets	1.00	1.76	2.32	2.69	2.95	3.15	3.28	3.35	3.40
	andp	1.00	1.76	2.26	2.66	3.00	3.23	3.68	3.60	3.60
	real	1.00	1.73	2.26	2.68	3.10	3.27	3.47	3.47	3.47
quicksort–600	subsets	1.00	1.80	2.41	2.84	3.15	3.38	3.53	3.64	3.71
	andp	1.00	1.75	2.25	2.75	3.20	3.34	3.79	3.97	4.00
	real	1.00	1.72	2.37	2.74	3.14	3.45	3.68	3.82	3.96
quicksort–750	subsets	1.00	1.78	2.36	2.75	3.04	3.25	3.38	3.47	3.53
	andp	1.00	1.71	2.42	2.60	3.13	3.55	3.66	3.75	3.67
	real	1.00	1.82	2.41	2.88	3.40	3.65	3.94	4.05	4.16
bpebpf–30	subsets	1.00	1.96	2.88	3.74	4.60	5.41	5.41	5.41	5.41
	andp	1.00	1.93	2.81	3.69	4.30	5.16	5.60	6.32	6.98
	real	1.00	1.83	2.44	3.66	4.40	4.40	5.50	5.50	7.33
bpesf–30	subsets	1.00	1.96	2.88	3.75	4.53	5.18	5.99	6.33	6.75
	andp	1.00	1.88	2.59	3.27	3.67	4.23	4.56	5.08	5.12
	real	1.00	1.80	2.57	3.60	4.50	4.50	4.50	6.00	6.00
pesf–30	subsets	1.00	1.47	1.74	1.92	2.05	2.14	2.20	2.26	2.31
	andp	1.00	1.41	1.65	1.83	1.95	2.02	2.10	2.18	2.26
	real	1.00	1.33	1.66	1.81	1.81	1.81	2.00	2.00	2.22

Table 6.4: Ideal and–parallelism

Program	Tracing System	Scheduling Algorithm	Processors								
			1	2	3	4	5	6	7	8	9
domino	Muse	subsets	1.00	1.95	2.88	3.75	3.92	3.92	3.92	3.92	3.92
		andp	1.00	1.89	2.74	3.56	3.92	3.92	3.92	3.92	3.92
	&–Prolog	subsets	1.00	1.98	2.94	3.86	4.75	5.61	6.42	7.20	7.97
		andp	1.00	1.98	2.92	3.86	4.78	5.61	6.54	7.32	8.26
	real		1.00	1.62	2.16	2.60	3.25	3.25	3.25	3.25	4.33
queens	Muse	subsets	1.00	1.91	2.72	3.41	3.41	3.41	3.41	3.41	3.41
		andp	1.00	1.87	2.54	3.41	3.41	3.41	3.41	3.41	3.41
	&–Prolog	subsets	1.00	1.97	2.92	3.82	4.70	5.48	6.22	6.93	7.55
		andp	1.00	1.95	2.77	3.77	4.72	5.33	5.89	6.30	6.48
	real		1.00	1.75	2.33	2.33	3.50	3.50	3.50	3.50	3.50
witt	Muse	subsets	1.00	1.12	1.17	1.19	1.19	1.19	1.19	1.19	1.19
		andp	1.00	1.08	1.12	1.12	1.12	1.12	1.12	1.12	1.12
	&–Prolog	subsets	1.00	1.05	1.07	1.08	1.09	1.09	1.09	1.09	1.09
		andp	1.00	1.05	1.07	1.08	1.09	1.09	1.09	1.09	1.09
	real		1.00	1.05	1.07	1.09	1.10	1.10	1.10	1.11	1.11
lanford1	Muse	subsets	1.00	1.95	2.83	3.62	4.20	4.62	4.62	4.62	4.62
		andp	1.00	1.89	2.67	3.37	4.32	4.62	4.62	4.62	4.62
	&–Prolog	subsets	1.00	1.98	2.91	3.79	4.59	5.34	6.04	6.67	7.45
		andp	1.00	1.97	2.92	3.82	4.73	5.53	6.27	7.29	8.09
	real		1.00	1.77	2.28	3.20	4.00	4.00	4.00	4.00	5.33
lanford2	Muse	subsets	1.00	1.85	2.55	3.15	3.73	4.30	4.77	5.12	5.50
		andp	1.00	1.91	2.50	2.94	4.02	4.51	5.51	5.85	5.88
	&–Prolog	subsets	1.00	1.99	2.99	3.98	4.97	5.95	6.92	7.88	8.85
		andp	1.00	1.99	2.98	3.97	4.96	5.91	6.88	7.87	8.85
	real		1.00	1.97	2.86	3.66	4.54	5.35	6.33	6.96	7.74

Table 6.5: Ideal or–parallelism

Chapter 7

Concurrency in Prolog Using Threads and a Shared Database

Chapter Summary

Concurrency in Logic Programming has received much attention in the past. One problem with many proposals, when applied to Prolog, is that they involve large modifications to the standard implementations, and/or the communication and synchronization facilities provided do not fit as naturally within the language model as we feel is possible. In this chapter we propose a new mechanism for implementing synchronization and communication for concurrency, based on atomic accesses to designated facts in the (shared) database. We argue that this model is comparatively easy to implement and harmonizes better than previous proposals within the Prolog control model and standard set of built-ins. We show how in the proposed model it is easy to express classical concurrency algorithms and to subsume other mechanisms such as Linda, variable-based communication, or classical parallelism-oriented primitives. We also report on an implementation of the model and provide performance and resource consumption data.

7.1 Introduction

Concurrency has been studied in the context of a wide range of programming paradigms, and many different mechanisms have been devised for expressing concurrent computations in procedural programming languages [BA82, Han77]. In fact, concurrency has also received much attention in the context of logic languages (see Chapter 2). However, most previous proposals have the drawback that they either involve large modifications to standard Prolog implementations, and/or the communication and synchronization facilities provided do not fit as naturally within the Prolog language model as we feel is possible. Although the approaches proposed to date are undoubtedly interesting and useful, we feel that they either do not provide all the features we perceive as most interesting in a practical concurrent Prolog system or they require complex implementations.

In this chapter we are interested in developing a model of concurrency for Prolog, whose main novelty is that both communication and synchronization are based on concurrent, atomic accesses to the shared Prolog database, which we argue can be used in the same way as a blackboard. We will show that, apart from the conceptual simplification, this choice creates very useful synergies in the overall language design, while remaining reasonably easy to implement. In our approach, an extension of the `assert/retract` family of Prolog calls allows suspension on calls and redos. We show that these primitives, when combined with the Prolog module system, have the same or richer functionality than blackboard-based systems, while fitting well within the Prolog model: they offer full unification instead of pattern matching on tuples and provide a clean interaction with Prolog control, naturally supporting backtracking. The model, as described in this paper, is available in the current distribution of CIAO (a next-generation, public domain Prolog system—see `http://www.clip.dia.fi.upm.es/Software`).

7.2 A First Level Interface

As mentioned before, the significant effort realized by the logic programming community in building parallel Prolog systems has proven that it is possible to construct sophisticated and very efficient multi-worker Prolog engines. However, it is also true that the inherent complication of these systems has prevented their availability as part of mainstream Prolog systems (with the possible exception of SICStus/MUSE, and, to a more limited extent, &-Prolog, Aurora, Andorra-I, and ACE). One of our main objectives in the design of the proposed concurrency model has been to simplify the low-level implementation, i.e., the modifications to the Prolog *engine* required, in order to make it as easy as possible to incorporate into an existing WAM-based Prolog system. Such simplicity should also result in added robustness.

With this in mind, we start by defining a set of basic building blocks for concurrency which we argue can be efficiently implemented with small effort. We will then stack higher-level functionality as abstractions over these basic building blocks.

7.2.1 Basic Thread Creation and Management

A thread corresponds conceptually to an independent Prolog evaluator, capable of executing a Prolog goal to completion in a local environment, i.e., unaffected by other threads. It is related to the notion of *agent* [HG91] or *worker* [Lus90, AK90b] used in parallel logic programming systems.

Thread creation is performed by calling eng_call/2, which is similar in spirit to the &/1 operators of &-Prolog, the spawn of MT-SICStus, or the new_engine of BinProlog. A call to eng_call(Goal, GoalHandle) first copies Goal and its arguments to the address space of a new thread and returns a *handle* in GoalHandle which allows the creating thread to have (restricted) control of and access to the state of the created thread. Execution of Goal then proceeds in the new thread within an independent environment. An exception may be raised if the spawning itself is not successful, but otherwise no further communication or synchronization with the caller occurs until a call to join_goal(GoalHandle) is made, unless explicitly programmed using the synchronization and communication primitives (Section 7.2.3).

A call to join_goal(GoalHandle) waits for the success or failure of the goal corresponding to GoalHandle. If a solution is found by the concurrent goal, this goal can at a later time be forced to backtrack and produce another solution (or fail) using backtrack_goal(GoalHandle). When no more solutions are needed from a given goal, the builtin release_goal(GoalHandle) must be called to release the corresponding thread. This also frees the memory areas used during the execution of the goal, and makes them available for other goals.

A number of specializations of eng_call/2 are useful in practice. A simplified version eng_call(Goal) causes Goal to be executed to completion (first solution or failure) in a new thread, which (conceptually) then dies silently. This behavior is useful when the created thread is to run completely detached from its parent, or when all the communication is performed using the communication / synchronization primitives (Section 7.2.3). There are also other primitives, such as kill_goal(GoalHandle), which kills the thread executing that goal and releases the memory areas taken up by it, which are useful in practice to handle exceptions and recover from errors.

7.2.2 Implementation Issues and Performance

The implementation requires the Prolog engine to be *reentrant*, i.e., several invocations of the engine code must be able to proceed concurrently with separate states. The modifications required are well understood from the parallel Prolog implementation work (we follow [HG91]).

A concurrent goal is launched by copying it with fresh, new variables, to the storage areas of a separate *engine*, which has its own working storage (*stack set*[1]) and attaching a thread (*agent*) to this stack set. The code area is shared and visible by all engines. Goal copying ensures that execution is completely local to the receiving WAM. This avoids many complications related to the concurrent binding and unbinding (on backtracking) of shared variables, since bindings/trailing, backtracking, and garbage collection are always local to a WAM, and thus need no changes with respect to the original, single-threaded implementation.

An important issue is how to handle goals which are suspended (e.g., are waiting at a *join*, or have executed other primitives which may cause suspension, to be described later) and goals which have returned a solution but are waiting to be joined. This issue was studied in [Her87, SH96]. There are two basic solutions: one is to *freeze* the corresponding stack set, which then cannot be used until the goal is resumed. The same occurs with a stack set containing the execution of a goal which has produced a solution but has pending alternatives. When this stack set is asked for another alternative a thread attaches to it and forces backtracking. This approach has the advantage of great simplicity at the cost of some memory consumption: it causes memory areas of the WAM (the upper parts of frozen stack sets) to be unused, since a new WAM is created for every new goal if the other WAMs are frozen and cannot be reused. WAMs are reused when a goal detaches after completion or when they are explicitly freed via a call to `release_goal/1`, in which case they are left empty and ready to execute another goal. The alternative is to reuse frozen stacks using *markers* to separate executions corresponding to different concurrent goals [Her87, SH96]. This can be more efficient in memory consumption, but is also more complicated to implement. We have implemented an intermediate approach which is possible if the stacks in the engine can be resized dynamically. We start with very small stacks which are expanded automatically as needed. This has allowed us to run quite large benchmarks with a considerable number of threads without running into memory exhaustion problems. It is also possible to shrink the stacks upon goal success, so that no memory is wasted in exchange for a small overhead. This is planned for future versions.

For our experiments we implemented the proposed primitives in the CIAO Prolog engine (essentially a simplified version of the &-Prolog engine, itself an independent evolution from SICStus 0.5-0.7, and whose performance is comparable to current SICStus versions running emulated code). We have used a minimal set of the POSIX thread primitives, in the hope of abstracting away the quite different management of threads provided in different operating systems, and to favor porting among UNIX flavors. All

[1]The memory areas used by a WAM, which are usually managed using a stack policy.

Thread creation	Engine coupling	Engine creation
2.03 ms / 702 LI	3.16 ms / 1091 LI	10.3 ms / 3579 LI

Table 7.1: Profile of engine and thread creation (average for 800 threads).

the experiments reported in this paper have been run on a SparcCenter 2000, with 10 55MHz processors, Solaris 2.5, CIAO-Prolog 0.9p75. All the measurements have been made using *walltime* clock.

Table 7.1 provides figures for several operations involving threads. Since the overhead per thread seems to remain fairly constant with the number of threads used, we show the average behavior for 800 (simultaneous) threads. Measurements correspond to the Prolog view of the execution: they reflect the time from a `eng_call(Goal)` is issued, to the time `Goal` is started. Times are given in ms. and, to abstract away from the processor speed, in "number of naive-reverse Logical Inferences" (at the ratio of 345 logical inferences per millisecond, the result given by `nrev` in the machine used). Although these numbers depend heavily on the implementation of O.S. primitives, we feel that providing them is interesting, since they are real indications of the cost of thread management.

The column labeled "Thread creation" reflects the time needed to start a thread, including the time used in copying the goal. The column labelled "Engine coupling" adds the time needed to locate an already created, free WAM, and to attach to it, and includes the initialization of the WAM registers. The column "Engine creation" takes into account the time used in actually creating a new engine (i.e., memory areas) and attaching to it. The last one is, as expected, larger, and this supports the idea of not disposing of the engines which are not being used. These figures are also useful in order to determine the threshold which should be used to decide whether execution should be sequential or parallel, based on granularity considerations [LGHD96]. Regarding memory consumption, the addition of thread support increased only very marginally the memory space needed per WAM.

A Note on Avoiding the Copy of the Calling Goal: Copying goals on launch, despite its advantages, may be very expensive. We support an additional set of primitives which perform sharing of arguments instead of copying.[2] To simplify the implementation and avoid a performance impact on sequential execution, concurrent accesses to the shared variables are not protected. The programmer has to ensure correct, locked accesses to them, including the effects of backtracking in other agents.

[2]That is, as long as the goals are being executed in the same machine (Section 7.3.5).

More complex management of variables can be built on top of these primitives by using, for example, attributed variables for automatic locking and publication of deterministic bindings, with techniques similar to those in [HCC95], and incremental, on demand copying of goal arguments, as shown in [CH96, LMSH97].

7.2.3 Synchronization and Communication Primitives

For the reasons argued previously, in this design we would like to use communication and synchronization primitives simpler to implement than those based on shared-variable instantiation. The use of the dynamic database that we propose as a concurrent shared repository of terms for communication and synchronization requires some (local) modifications to the semantics of the accesses to the dynamic database, but also results in some very interesting synergies.

7.2.4 Making the Database Concurrent

We start by assuming that we can mark certain dynamic predicates as concurrent by using a `concurrent/1` declaration. The implication is that these predicates can be updated concurrently and atomically by different threads. We also assume for simplicity that these predicates will only contain facts, i.e., they are `data/1` predicates in the sense of [BCHP96] and Ciao (this makes them faster and helps global analysis). Finally, we assume that if a concurrent predicate is called and no matching fact exists at that time in the database, then the calling thread *suspends* and is resumed only when such a matching fact appears (i.e., is asserted by a different thread, instead of failing). With these assumptions, there is a relationship between the Linda primitives (Table 7.2, middle column) and the Prolog *assert/retract/call* family of builtins in the context of concurrent predicates (right column). The first three Linda operations, `out/read/in`, have now clear counterparts in terms both of information sharing and synchronization. In the following example,

Operation	**Linda**	`concurrent/1` **Facts**
Put tuple	`out`	`asserta/1, assertz/1`
Wait, read tuple	`read`	`call/1`, simple call
Wait, read, delete tuple	`in`	`retract/1`
Read tuple or fail	`read_noblock`	`call_nb/1` (+)
Read and delete or fail	`in_noblock`	`retract_nb/1` (+)
No more tuples	—	`close_predicate/1`
(More tuples may appear)	—	(`open_predicate/1`)

Table 7.2: Comparing Linda primitives and database-related Prolog primitives

```
:- concurrent state/1.

p:-
    eng_call(q),
    state(X),
    !,
    (
        X = failed ->
        ...
    ;
        ...
    ).
```

```
q:-
    <...produce Result...>,
    !,
    asserta(state(correct(Result))).
q:- asserta(state(failed)).
```

`p` launches predicate `q` and waits for notification of its final state, which may be a `Result` or a *failed* state (the use of the Prolog cuts will be clarified further later). Making the dynamic predicate `state/1` be `concurrent` ensures atomic updates and the suspension of the call to `state(X)` in `p`.

One interesting difference with Linda primitives appears at this point: it is clear that we may want to be able to backtrack into a call to a concurrent predicate (such as the one to `state(X)` above). The behavior on backtracking of calls to concurrent predicates is as follows: if an alternative unifying fact exists in the database, then the call matches with it and proceeds forward again. If no such fact exists, then execution suspends until one is asserted. This is the natural extension of the behavior when the predicate is called the first time, and makes sense in our concurrent environment where facts of this predicate can be generated by another thread and may appear at any time. It allows, for example, implementing producer-consumer relations using simple failure-driven loops. In the following temperature example a thread accesses a device for making temperature readings, and asserts these, while a concurrent reader accesses them in a failure driven loop as they become available.

```
:- concurrent temp/1.

temperature:-
    eng_call(read_temp),
    produce_temp.

produce_temp:-
    (
        read_temp_device(Temp) ->
        assertz(temp(Temp)),
        produce_temp
    ;
        assertz(temp(end))
    ).
```

```
read_temp:-
    temp(Temp),
    (
        Temp = end ->
        true
    ;
        <...work with Temp...>,
        fail
    ).
```

When no more temperature readings are possible, `read_temp_device/1` fails and the `end` token, instead of a a temperature, is stored, which causes the reader to exit. Conceptually, when backtracking is performed, the *next clause pointer* moves downwards in the clauses of the `temp/1` predicate until the last fact is reached. Then, the calling thread waits for more facts to appear. Note that assertion is done using `assertz/1`, which adds new clauses at the end of the predicate, so that they can be seen by the reader waiting for them. If `asserta/1` were used, newly added facts would not be visible, and thus the reader would not wake up and read the new data available. Also, note that the data produced remains, so that other readers could process it as well by backtracking over it. For example, assume the temperature asserted has the time of the reading associated with it. Different readers can then to consult the temperature at a given time concurrently, suspending if the temperature for the desired time has not been posted yet. Alternatively, if the call to `temp(Temp)` above is replaced with a call to `retract(temp(Temp))`, then each consumer will eliminate a piece of data which will then not be seen by the other consumers. This is useful for example for implementing a task scheduler, where consumers "steal" a task which will then not be performed by others.

The concurrent database thus allows representing a changing outside world in a way that is similar to other recent proposals in computational logic, such as condition-action rules [Kow96]. A sequence of external states can be represented by a predicate to which a series of suitably timestamped facts are added monotonically, as in the temperature sensor example above. Processes can sense this state and react to it or suspend waiting for a given outside event to happen.

One nice characteristic of the approach, apart from naturally supporting backtracking, is that many concurrent programs using shared facts are very similar to the non-concurrent ones. There is, however, a subtle difference which must be taken into account: when calling standard, non-concurrent facts with alternatives, the choicepoint disappears when the last fact is accessed. In the reader the "last fact" was assumed to be that with the "`end`" token, but this did not make the choicepoint pushed in by the access to the database go away. A failure at a later point of the reader would cause it to backtrack to this choicepoint, and probably suspend — which may or may not be desired. Getting around this behavior is possible by simply putting an explicit cut at the point in which we decide that no more facts are needed (i.e., the communication channel has been conceptually closed), so that the dynamic concurrent choicepoint is removed. This is the reason for the cut in the first example of synchronization: we just wanted to wait for a fact to be present, and then we did not want to leave the choicepoint lying around.

7.2.5 Closing Concurrent Predicates

There are cases in which we prefer failure instead of suspension, if no matching is possible. This can be achieved in two ways. The first one is using non-blocking (`_nb`) versions of the retract and call primitives (marked + in Table 7.2), which fail instead of suspending, while still ensuring atomic accesses and updates. The second, and more interesting one, is explicitly *closing* the predicate using `close_predicate/1`. This states that all alternatives for the predicate have been produced, and any reader backtracking over the *last* asserted fact will then fail rather than suspending. The example can now be coded as:

```
:- concurrent temp/1.

temperature:-
    eng_call(read_temp),
    produce_temp.
```

```
produce_temp:-                          read_temp:-
    (                                       temp(Temp),
        read_temp_device(Temp) ->           <...work with Temp...>,
         assertz(temp(Temp)),               fail.
         produce_temp                   read_temp.
    ;
        close_predicate(temp/1)
    ).
```

where the call to `temp(Temp)` eventually fails after the predicate is closed. This is useful for example for marking that a stream modeled by a concurrent predicate is closed: all the threads reading/consuming facts from this predicate will fail upon the end of the data. For completeness, a symmetrical `open_predicate/1` call is available in order to make a closed predicate behave concurrently again (although it is arguably best practice not to re-open closed predicates).

7.2.6 Local Concurrent Predicates

New, concurrent predicates can be created dynamically by calling the builtin `concurrent/1`. The argument to `concurrent/1` can be a new predicate. Also, if the argument of the call to `concurrent/1` contains a variable in the predicate name, the system will create dynamically a new, local predicate name. This allows *encapsulating* the communication, which is now private to those threads having access to the variable:

```
temp:-
    concurrent(T/1),
    eng_call(read_temp(T)),
    produce_temp(T).

produce_temp(T):-                       read_temp(T):-
    (                                       T(Temp),
        read_temp_device(Temp) ->           <...work with Temp...>,
        assertz(T(Temp)),                   fail.
        produce_temp                    read_temp(_).
    ;
        close_predicate(T/1)
    ).
```

where we could replace the higher-order syntax `T(X)` supported by CIAO Prolog with calls to `=..` and `call/n` (e.g., `call(T,Data)`). Note that the functionality at this point is not unlike that of a *port*, but with a richer backtracking behavior.

Another way of encapsulating communication stems from an interesting synergy between the concurrent database and the module system. Concurrent predicates are, as usual, in principle local to the module in which they appear. If they are not exported, they constitute is a channel which is local to the module and can only be used by the predicates in it. The module-local database thus acts as a local blackboard. By exporting and reexporting concurrent predicates between modules, separate, *private blackboards*, can be easily created whose accessibility is restricted to those importing the corresponding module. This is particularly useful when several instantiations (objects) are created from a given module (class) – see [PH99].

7.2.7 Logical View vs. Immediate Update

A final difference between concurrent predicates and standard dynamic predicates is that the logical view of database updates [LO87], while convenient for many reasons, is not really appropriate for them. In fact, if this view were implemented then consumers would not see the facts produced by sibling producers. Thus, an immediate update view is implemented for concurrent predicates so that changes are immediately visible to all threads.

7.2.8 Locks on Atoms/Predicate Names

A method for associating semaphores [BA82] to atoms / predicate names is available. Mimicking those in procedural languages, a counter is associated with each atom which can be tested or set atomically using `atom_lock_state(+Atom, ?Value)`. It can be atomically tested and decremented if its value is non-zero, or waited on if it is zero, using `lock_atom(+Atom)`, and incremented atomically using `unlock_atom(+Atom)`. The implementation is very cheap, avoiding the overkill of simulating semaphores with concurrent predicates, when only a simple means of synchronization is needed.

7.2.9 Implementation Issues and Performance

Concurrent accesses are made atomic by using internal, user-transparent locks, one per predicate. Every call to a concurrent predicate creates a dynamic choicepoint with special fields. In particular, its *next alternative* field points to the next clause to try on backtracking through an indirect pointer. All the indirect pointers from different choicepoints leading to a given clause are linked together into a chain reachable from that clause, so

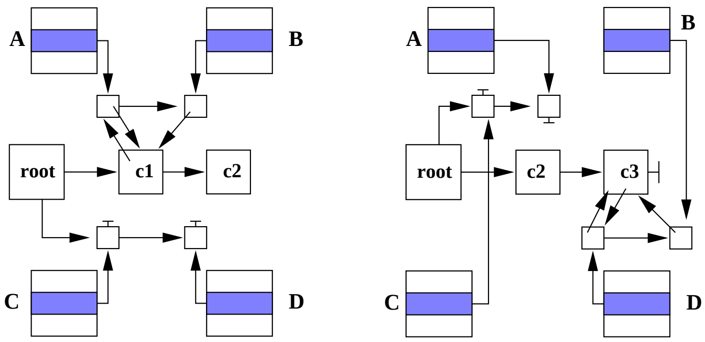


Figure 7.1: Choicepoints and suspended calls before and after updating clauses

that any goal updating the predicate can access and relocate all of them atomically if needed (for example, if the clause is removed). Calls which suspend do not have their associated choicepoint removed, and the corresponding indirect pointers are linked in a separate suspension chain (Figure 7.1, left). When a thread tries to access its next alternative and no alternative matching clause exists, the thread waits on changes to that indirect pointer instead of failing. This behavior ultimately depends on whether the call was blocking or not, and on whether the predicate was closed or not at the time. We discuss the interesting case of blocking calls on open predicates.

When a clause is removed, the chain of indirect pointers leading to it is checked: some of the pointers might be moved forwards in the clause list to the next possibly matching instance (as dictated by indexing), and in some cases it can be determined that no matching clause exists. In the latter case, they are linked to the chain of suspended calls. On the other hand, every time a new clause is appended, the list of suspended calls is checked, and those which may match the new clause (again, according to indexing), are made to point to that new clause. This is performed even if the affected goal is not actively waiting on an update of the clause, but executing some other code.

Figure 7.1 depicts a possible state of the database before and after some clause updates take place. On the left, choicepoints **A** and **B** point (indirectly) to clause `c1` as next clause to try. Choicepoints **C** and **D** point to *null* clauses, and they are either suspended, or they would suspend should they backtrack now. Let us remove `c1` and add `c3`. The thread which adds / removes clauses is in charge of updating the affected pointers, based on indexing considerations. The call from **A**, which was pointing to `c1`, does not match neither `c2` nor `c3`, so it points to the *null* clause now, and is enqueued in the list of calls to suspend. The call from **B** may match `c3` but does not match `c2`, and its indirect pointer is set accordingly. The call from **C** does not match `c3` and so its state does not change. And, finally, the call from **D** might match `c3`, and it is updated to point to this clause.

Primes	Conc	Data
5000	1511	1915
10000	2475	5204
15000	4775	9100
20000	6386	12560
25000	9061	17804
30000	11900	24298
35000	13252	29450

Table 7.3: Sieve of Erathostenes

Fact spec	Memory	bytes/fact	bytes/arg
p/0	1264	21.57	—
p/1	1753	29.91	8.34
p/2	1871	31.93	5.18
p/4	2105	35.92	3.58
p/8	2571	43.87	2.78
p/16	3507	59.85	2.39
p(g/1)	1615	27.56	5.99
p(g/2)	1753	29.91	4.17
p(g/4)	1967	33.57	3.00
p(g/8)	2435	41.86	2.53
p(g/16)	3373	57.56	2.24

Table 7.4: Memory usage, 60000 facts

The cut needs some additional machinery to retain its semantics. Not only the (dynamic) choicepoints in the scope of a cut should be swept away (which boils down to updating a pointer), but also the possibly suspended goals corresponding to the concurrent choicepoints must be removed. This is currently done by traversing part of the choicepoint stack, following the links to suspended calls, and removing them.

The implementation of concurrent predicates is not trivial, but we argue that it is much simpler than implementing variable-based communication that behaves well on backtracking. Also, it affects only one part of the abstract machine, database access, which is typically well isolated from the rest. In our experience, the changes to be performed are fairly local. The resulting communication among threads based on access to the database may be slower than communication using shared variables, although, depending on the implementation, reading can be faster. However, note that in this design we are not primarily interested in speed, but rather in flexibility and robustness, for which we believe the proposed solution is quite appealing. Also, in the proposed implementation the execution speed of sequential code which makes no use of concurrency is not affected in any way, which is not as easy with a shared-variable approach. Furthermore, the fact that concurrent predicates should not meet the logical view of database updates [LO87], eliminates the need to check whether a fact is alive or not within the time window of a call, which makes, in some cases, the access and modification of concurrent predicates up to more than twice as fast as that of standard dynamic predicates.

As an example of the impact on speed of the immediate database updates, Table 7.3

1	2	4	6	12
4327	2823	1687	1400	1625

Table 7.5: Adding and removing facts from a database, 10 processors available

shows timings (in milliseconds) for a database implementation of the well-known Sieve of Erathostenes, using a failure loop to both traverse the table of live elements and to remove multiples. The “Data” column corresponds to the version which uses the CIAO `data/1` declaration (which is faster than `dynamic`, and specialized for facts). The “Conc” row uses the `concurrent/1` declaration. Clause liveness (i.e., whether a given clause should or not be seen by a given call) must be tested quite often in this case, which accounts for the performance jump. On the other hand, other patterns of accesses to database perform this liveness test quite sparingly (if at all), and benefit less from the immediate update, suffering instead from the mandatory lock of the predicates being accessed. However, the factors seem to compensate even in the worst cases since we have not been able to find noticeable slowdowns.

With respect to memory consumption, Table 7.4 lists average memory usage per fact and per argument for the CIAO Prolog implementation in a benchmark which asserts 60000 facts to the database. A fact `p` with different arguments (integers) was asserted, as well as a fact with a single argument, containing a functor with different numbers of arguments (integers again). It is encouraging that these figures are well behaved, as we may expect large numbers of facts asserted in the database.

Another interesting issue is the impact of contention in concurrent predicate accesses. Our implementation ensures that concurrent accesses to different predicates will not interfere with each other: Table 7.5 shows speeds for the access and removal of a total of 10000 facts using different numbers of threads. Each thread accesses a different predicate name, which results in speedups until the number of threads is greater than the number of available processors, when other contention factors appear. However, there is obviously some interference in the concurrent accesses to the same predicate.

7.3 Some Applications and Examples

We now illustrate the use of the proposed concurrency scheme with some examples.

7.3.1 Implementing Condition-Action Rules

The concurrent database also has the advantage that it allows representing a changing outside world in a way that is similar to that of condition-action rules [Kow96]. A sequence of external states can be represented by an open predicate to which a series of suitably timestamped facts are added monotonically, as for example data from a temperature sensor. Processes can sense this state and react to it or suspend waiting for a given outside event to happen. As an example, consider the implementation of condition-action rules shown in Figure 7.2. On the left the original source code is shown, which includes the library file `c_a_rules`. This file adds, in Ciao Prolog style, local declarations for operators and predicates to perform the translation. On the right a possible translation is shown. `initialization/1` is an ISO directive that instructs the runtime system to start the argument goal on startup. In this case, the monitor for the rule is started as a (perpetual) daemon, and it checks two concurrent facts to decide wether or not to turn off a heater—this is also represented as a concurrent fact, which might in turn be consulted by another CA rule.

```
:- module(someactions,[]).
:- use_module(htrctrl,[heater/1]).
:- use_module(sensors,[temp/1]).

:- use_package(c_a_rules).

  {temp(X), heater(on)},
   X > 5 => heater := off.

...
```

```
:- module(someactions,[]).
:- use_module(htrctrl,[heater/1]).
:- use_module(sensors,[temp/1]).

:- initialization eng_call(cond_act_rule_1).

cond_act_rule_1:-
    current_fact(temp(X)),
    current_fact(heater(on)),
    X > 5,
    turnheater(off),
    cond_act_rule_1.

turnheater(NewState) :-
    retract_fact(heater(OldState)),
    asserta_fact(heater(NewState)).
```

Figure 7.2: Compiling condition-action rules (left) to Ciao code (right)

7.3.2 Semaphores

It should be noted that other abstractions, such as general semaphores, are also straightforward to implement. As an example, Figure 7.3 shows the implementation of a general semaphore for concurrency. The state of the semaphore is determined by the number of facts of the predicate. This example also hilights how a concurrent fact (in this case of arity zero) can be created dynamically. This ability to create on-the-fly (the equivalent of) dynamic semaphores is of great help, e.g., if database-based O'Ciao objects [PH99] are to be extended to the concurrent case, a lock has to be created per object.[3]

```
init(Mutex, Value):-
    concurrent(Mutex/0),
    forall(1, Value, asserta(Mutex)).

signal(Mutex):- asserta_fact(Mutex).
wait(Mutex):- retract_fact(Mutex).
```

Figure 7.3: A general semaphore using concurrent facts

7.3.3 The Five Dining Philosophers

Figure 7.4 presents the code for the problem of the Five Dining Philosophers, with the aim of showing how a standard solution can be adapted to the concurrent database approach. The code mimics the solution presented in [BA82]. Each philosopher is modeled as a concurrent goal which receives its number as an argument. Fork-related actions are modeled by accesses to a concurrent predicate `fork/1`. A global semaphore, associated with the atom `room`, controls the maximum number of philosophers in the dining room, and also makes sure that all philosophers start at once.[4] No attempt is made to record when a philosopher is thinking or eating, but this can be done by asserting a concurrent predicate recording what every philosopher is doing at each time.

7.3.4 A Skeleton for a Server

A *server* is a perpetual process which receives requests from other programs (clients) and attends them. Typically, the server should accept more queries while previous ones are being serviced, since otherwise the service would stop temporarily. Therefore, servers

[3] Ciao Prolog has actually locks associated to atoms, which are presumably better suited for this task. However, we want to note that the mechanisms seen so far are enough to achieve the results we seek.

[4] Actually, this is not strictly needed: letting philosophers think and eat as they become alive does not change the behavior of the algorithm, but this decision illustrates the use of atom-based locks for global synchronization.

```
:- concurrent fork/1.

philosophers:-
    atom_lock_state(room, 0),
    eng_call(philosopher(1)),
    eng_call(philosopher(2)),
    eng_call(philosopher(3)),
    eng_call(philosopher(4)),
    eng_call(philosopher(5)),
    atom_lock_state(room, 4).

eat:- ...
think:- ...

fork(1). fork(2).
fork(3). fork(4). fork(5).

philosopher(ForkLeft):-
    ForkRight is (ForkLeft mod 5) + 1,
    think,
    lock_atom(room),
    retract(fork(ForkLeft)),
    retract(fork(ForkRight)), !,
    eat,
    assertz(fork(ForkLeft)),
    assertz(fork(ForkRight)),
    unlock_atom(room),
    philosopher(ForkLeft).
```

Figure 7.4: Code for the Five Dining Philosophers

```
main:- catch(server, _AnyError, halt).

server:-
        wait_for_request(Query),
        eng_call(service(Query)),
        server.
```

Figure 7.5: A skeleton for a server

usually are multithreaded, and children fork from the parent in order to handle individual requests. A simple skeleton for a server is shown in Figure 7.5. The main thread waits for a request and, when one arrives, launches a child thread to process it. The server itself is started within the context of a catch/throw construction which will exit the execution should the server receive any external signal.[5]

Possible internal errors of the server can be dealt with by the `service/1` predicate itself, since each one of its invocations is detached from the main thread. The shared database

[5] Exceptions in CIAO Prolog are installed on a per-thread basis, so every concurrent goal can have its own exception handlers without altering the behavior of the other threads.

	Start thread	Gather bindings
No handle, local	`..., G &, ...`	—
Handle, local	`..., G &> H, ...`	`H <&`
Remotely concurrent	`(G &) @ S`	—
Locally concurrent, remote execution	`(G @ S) &`	—
Remote handle, remotely concurrent	`(G &> H) @ S`	`(H <&) @ S`
Local handle, remote execution	`(G @ S) &> H`	`H <&`

Table 7.6: Starting concurrent / distributed goals and waiting for bindings

provides a communication means in case the children have to report any data to the dispatcher.

7.3.5 Implementing Higher-Level Concurrency Primitives

The interface offered by the primitives related to threads, locks, and database is sufficient for building many different concurrent programs, but it is somewhat low-level. For example, the number of simultaneous threads has to be controlled explicitly as part of the application code. Similarly, waiting for completion of the computation of a thread and accessing the bindings created by it need the execution of a (fixed) sequence of steps. Also, implementing backtracking over concurrent goals requires some often repeated coding sequences. Such sequences are clear candidates to be abstracted as higher-level constructs.

Using the basic primitives, we have implemented the set of concurrency and distributed execution constructs proposed in [HCC95, CH96], some examples of which are shown in table 7.6. Remote goals are executed in a server `S`, specified with the placement operator `@/2` (so that, for example, `G @ S` means "execute `G` at `S`, wait for its completion, and import the bindings performed"). Handles (`H`) allow waiting for the (remote) completion of the goal, and gathering the bindings. Lack of space prevents us from including the actual implementation code, but it is easy to port the implementations given in [HCC95, CH96]. Using concurrent predicates instead of the external blackboard used there results in a simplification of the code. Significant simplifications also stem from the fact that with the proposed primitives goals which have produced a solution can be left frozen and then asked for additional solutions. Thus, concurrent and distributed goals now need not be called in the context of `findall`.

Processors	Granularity level								
	17	18	19	20	21	22	23	24	25
1	1289	1253	1253	1287	1266	1264	1285	1305	1309
3	463	455	480	491	524	623	533	820	1309
5	287	290	312	312	376	319	510	818	1309
7	219	220	210	244	309	318	504	823	1309
9	178	178	197	220	213	330	519	836	1309

Table 7.7: Deterministic and-parallel scheduler: granularity against no. of processors

As a simple example, we discuss the implementation of a version of the traditional &-Prolog `&/2` operator, which, placed instead of a comma, specifies that the two adjacent goals are to be executed in parallel and independently: `GoalA & GoalB`. This operator was implemented at a very low-level (i.e., modifying the underlying abstract machine) in the &-Prolog system [HG91] and in other systems [GHPSC94], which resulted in very good performance, but at the cost of a non-trivial amount of implementation work. Figure 7.6 shows the code for our source-level implementation which assumes that the goals to be executed are deterministic. Extending it for nondeterministic goals is easy, but makes the code too long for our space limitations. However, we will compare performance results for both the simple implementation and the one which fully supports backtracking.

In this implementation, the parallel operator `&/2` assigns a unique identifier to every parallel conjunction. One of the parallel goals is executed locally, while the other is stored in the database, together with its identifier, waiting for a scheduler to pick it up. Such a scheduler is implemented by the predicate `scheduler/0`. To use N processors of a parallel machine, $N-1$ threads should be created, all running initially `scheduler/0`. As soon as one goal is posted to the database, one of the threads running the scheduler grabs and executes it, leaves the solution in the database, and fails in order to wait for another goal. If no free schedulers are available, the main thread may find, upon completion of the local goal, that the goal stored in the database is still there. Then, this local thread picks it up and executes it locally. On the other hand, if the solution waited for is not in the database, and the goal left there from the conjunction has been taken, the main thread switches personality and tries to execute any other goals present in the database while also checking whether the solution it requires for the original goal has been posted or not.

```
:- concurrent goal_to_execute/2.
:- concurrent solution/3.

GoalA & GoalB :-
   new_id(IdA),
   assertz(goal_to_execute(IdA, GoalA)),
   call_with_result(GoalB, ResultB),
   (
      retract_nb(goal_to_execute(IdA, GoalA)) ->
      call_with_result(GoalA, ResultA)
      ;
      repeat,
      perform_some_other_work(IdA, GoalA, ResultA), !
   ),
   ResultA = success,
   ResultB = success.

perform_some_other_work(Id, Sol, Res):-
   retract_nb(solution(Id, Sol, Res)).
perform_some_other_work(_Id, _Sols, _Result):-
   retract_nb(goal_to_execute(Id, Goal)), !,
   call_with_result(Goal, Result),
   assertz(solution(Id, Goal, Result)),
   fail.

scheduler:-
   retract(goal_to_execute(Id, Goal)),
   call_with_result(Goal, Result),
   assertz(solution(Id, Goal, Result)),
   fail.

call_with_result( Goal, success) :- call(Goal), !.
call_with_result(_Goal, failure).
```

Figure 7.6: Code for an and-parallel scheduler for deterministic goals

Processors	Granularity level								
	17	18	19	20	21	22	23	24	25
1	1621	1555	1530	1549	1541	1582	1584	1613	1325
3	571	570	570	588	611	746	635	1062	1332
5	367	363	380	392	453	411	617	1029	1332
7	286	290	266	299	378	393	628	1081	1332
9	246	229	247	256	262	403	633	1040	1332

Table 7.8: Non deterministic and-parallel scheduler: granularity against no. of processors

This very naive implementation cannot, of course, achieve the same performance as &-Prolog (and this is obviously not the objective of the exercise). However, it is interesting that a correct selection of the granularity level [LGHD96] does produce speedups due to parallel execution on at least some benchmarks. Table 7.7 shows times (in milliseconds) for the parallel execution of the doubly recursive Fibonacci benchmark (computing the 24^{th} Fibonacci number) using the scheduler for deterministic goals. Each column is labelled with a different granularity level, i.e., the column labeled "17" corresponds to a call which stops spawning goals from the call to compute the 17^{th} Fibonacci number downwards. Table 7.8 shows results for the same benchmark using a scheduler which supports non-deterministic goals. The lower the granularity level, the more goals are executed in parallel, and the smaller they are. The speedups shown approach linearity when execution is performed at a large enough granularity level. As expected, execution also speeds up as more parallel goals are available, until a turning point is reached (at the level of granularity of 17). At this level of granularity the cost of accessing the database for copying goals and recovering the solutions exceeds the speedup obtained from parallel execution. The nondeterministic scheduler, additionally, adds an overhead to the execution, which for this benchmark case ranges from 16% to 30%, with an isolated peak of 39%—and therefore, has a higher granularity, with the "turning point" in 18.

7.4 Conclusions

We have proposed a new model to express concurrency, including communication and synchronizations. It inherits, in some ways, from proposals like Linda, but it differs from them in that the language semantics is widened to allow for the new synchronization capabilities.

The proposal is expressive enough as to base in it several well known synchronization methods. The implementation of the thread creation, based on copying goals to a separate environment, allows taking advantage of most of the already developed technology for sequential execution.

As future work, besides improving the efficiency of the system, we plan to investigate the use of the proposed low-level techniques as a basis for implementing higher-level constructions. Along these lines, we are currently working on a library which implements remote concurrent objects.

Chapter 8

Conclusions

Since every chapter has its own conclusions, we will just summarize them here, and try to make clear again the relationships among the goals achieved in the thesis.

8.1 Parallelism

The use of parallelism is one of the initial approaches aimed at improving the efficiency in Logic Programming, as we commented in Chapter 2. Unfortunately, the associated implementation techniques are notably complex, and adopting some optimizations is more difficult than in the sequential case. Among the associated problems we can cite the increased memory usage and some speed-down relative to an ideal parallel execution due to the need of creating tasks with the proper format for parallel execution. We have studied these two problems and worked on an implementation of a simulation tool to analyze *post-mortem* the behavior of a parallel execution:

- We have studied how the memory usage associated to the parallel backtracking can be diminished. We have found a method, based on a slight modification of the RAP-WAM, which achieves substantial memory savings in the case of deterministic predicates, and we have seen its usefulness in several test cases.

- We have also evaluated the speed-down for the parallel execution of a special class of recursive predicates which transform a data structure into a similar one, or which aggregate elements of a data structures. We have studied a solution based on program rewriting which achieves important improvements in the speedup of the programs while keeping their original semantics.

- The simulation tool allows comparing a real execution with an ideal execution on a predetermined number of processors, or finding out the minimum number of processors needed in order to obtain the maximum speedup in an execution. The first case is useful, e.g., to compare several scheduling algorithms, and the second to verify whether an upgrade in machine hardware could result in increased speed, as well as to study the change in the amount of parallelism actually exploited in a program when the number of processors is increased. We have evaluated an implementation of such a tool, and we have seen that their results are very similar to those obtained by direct experimentation.

8.2 Visualization

A well-designed visualization gives an intuitive description of program executions, and leads to a better understanding of this execution. This is specially relevant in parallel and constraint programs, due to the complexity of their execution, which can be completely understood only by knowing the details of the inner architecture of the system. In most cases different code for a program can give very different execution times, although the program semantics does not change. A number of visualization tools have been designed and implemented:

- *APT*, a tool to visualize sequential execution which can be used to study behavior patterns of small and medium sized executions, although its main application field is probably education.

- *VisAndOr*, which focuses on performance study for parallel programs, featuring a tailored depiction. Its usefulness has been shown in several cases, some of which have been showcased in this thesis. *VisAndOr* and other related tools are instrumental in the optimization and debugging of parallel systems, and in the study of the behavior of parallel logic programs.

- *VIFID* and *TRIFID*, which show a data-oriented view of the data in constraint logic programming over finite domains. This is in contrast with the previous two tools, which aim at representing control-related issues.

8.3 Concurrency

Concurrency, as a means to better use computer resources and to realize interactive and reactive systems, is necessary in any current programming language. In this thesis, we have put forward a new proposal of concurrency based on integrating into Prolog of a programming style *à la Linda* by augmenting the semantics of a designated set of (dynamic) predicates. This proposal can be used as a base to implement and study other types of constructs for concurrency and distribution at a higher level.

8.4 Future Work

We will point out here certain issues which remain open and which can continue the work of the present thesis:

Integration of optimizations in parallel models: although several programming systems do have parallel programming capabilities, none of them incorporates the different optimizations which have been proposed in several papers. Integrating and implementing all of them is a non trivial pending work. Part of its difficulty comes from the following reasons:

- The implementation in itself is complex and delicate.
- The interactions among the several types of optimizations is not well studied, and the requirements of each of these optimizations can be not only different, but incompatible.

We propose a study of the different optimizations for sequential and parallel programming, and of the conditions needed for each of them. A similar study for the sequential case has recently been initiated [DN00a], departing from previous comparisons in using as base the same execution engine, therefore not contaminating the performance results for several optimizations with other design decisions already been made.

Analysis environment addressed to the optimization of parallel implementations: it seems there is a lack of practical results (in the sense of integrated programs) in the wide and quite successful field of program analysis. One of the most promising systems was the *&–Prolog* compiler [MBdlBH99], able to extract independence information in a user-transparent fashion. Meanwhile, other analysis for determinism, granularity, etc. have been developed. Putting all of them together in a similar environment (similar, e.g., to CiaoPP [HBPLG99]) would allow obtaining many information of outstanding importance for the optimization of parallel, distributed, and sequential execution.

On-Line simulation: scheduling algorithms face a problem: very often its efficiency differs according to the goal / program being executed. Moreover, systems with and-

and or-parallelism have different scheduling needs according to the type of parallelism in each moment: it does not seem to be wise to use the same scheduling strategy in both execution models. It would be possible to gather the information pertaining the execution profile and perform an online simulation which helps to create a better scheduling in new executions of the same program. A similar task is already been performed by some constraint languages [COS96] in order to make more intelligent decisions in the process of labeling.

Integrating debugging-oriented analysis with program visualization: there are currently analyzers able to test high-level specifications against a program, and which generate code to check at run-time these properties which have not been proved (or disproved) at compile-time. This code can be associated to calls/messages handled by a visual debugger, which would be triggered at the moment a property is violated.

Designing and implementing higher-level concurrency constructs: we want to continue with the study of explicit concurrency, and explore the possibilities of expressing creation and synchronization of tasks at a higher level, using the basic building blocks shown here. These proposals are, as an important property, based on separate computation spaces, which allow simplifying the implementation. The absence of implicit communication allows unifying in the same proposal distributed and concurrent execution.

Adapting analysis techniques to concurrency: the analysis of concurrent systems is difficult. However, the proposals of *restricted* concurrency (in the sense that there is a lack of implicit communication) we have made, can lead to an easier analysis, since the sequential parts and the synchronization points are implicit in the program code. Studying whether these concurrency mechanism are advantageous or not regarding analyses is future work.

Bibliography

[ABB93] Henrik Arro, Jonas Barklund, and Johan Bevemyr. Parallel bounded quantification—preliminary results. *ACM SIGPLAN Notices*, 28:117–124, 1993.

[Abr96] J-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[ABV00] R. T. Alexander, J. M. Bieman, and J. Viega. Coping with java programming stress. *IEEE Computer*, 33(4), April 2000.

[Adl95] Richard M. Adler. Emerging standards for component software. *IEEE Computer*, 28(3):68–77, March 1995.

[AK90a] K. A. M. Ali and R. Karlsson. Full Prolog and Scheduling Or-parallelism in Muse. *International Journal of Parallel Programming*, 1990. Vol. 19, No. 6, pp. 445–475.

[AK90b] K. A. M. Ali and R. Karlsson. The Muse Or-Parallel Prolog Model and its Performance. In *1990 North American Conference on Logic Programming*, pages 757–776. MIT Press, October 1990.

[AK91] Hassan Ait-Kaci. *Warren's Abstract Machine, A Tutorial Reconstruction*. MIT Press, 1991.

[AKK+92] Seiichi Aikawa, Mayumi Kamiko, Hideyuki Kubo, Fumiko Matsuzawa, and Takashi Chikayama. Paragraph: A Graphical Tuning Tool for Multiprocessor Systems. In *Proceedings of the Intl. Conf. on Fifth Generation Computer Systems*, pages 286–293. Tokio, ICOT, June 1992.

[Ali88] K. A. M. Ali. Or-parallel Execution of Prolog on the BC-Machine. In *Fifth International Conference and Symposium on Logic Programming*, pages 253–268, Seattle,Washington, 1988. MIT Press.

[AM94] K. R. Apt and E. Marchiori. Reasoning about Prolog programs: from modes through types to assertions. *Formal Aspects of Computing*, 6(6):743–765, 1994.

[And91] G. R. Andrews. *Concurrent Programming: Principles and Practice*. Benjamin/Cummings, 1991.

[AR97] L. Araujo and J.J. Ruz. A Parallel Prolog System for Distributed Memory. *Journal of Logic Programming*, 33(1):49–79, October 1997.

[AS89] G.R. Andrews and F.B. Schneider. Concepts and notations for concurrent programming. In N. Gehani and A.D. McGettrick, editors, *Concurrent Programming*. Addison-Wesley, 1989.

[AVWW96] J. Armstrong, R. Virding, C. Wistrom, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall, 1996.

[BA82] M. Ben-Ari. *Principles of Concurrent Programming*. Prentice Hall International, 1982.

[Bar90] Jonas Barklund. *Parallel Unification*. PhD thesis, Comp. Sci. Dept., Uppsala Univ., Uppsala, 1990.

[Bar94] Jonas Barklund. Bounded quantifications for iteration and concurrency in logic programming. *New Generation Computing*, 12:161–182, 1994.

[BBM99] P. Behm, P. Benoit, and J.M. Meynadier. Meteor: A Successful Application of B in a Large Project. In *FM 99 — World Conference on Formal Methods in the Development of Computing Systems*, number 1708 in LNCS, pages 369–387. Springer Verlag, 1999.

[BC91] A. Brogi and P. Ciancarini. The Concurrent Language, Shared Prolog. *ACM Transactions on Programming Languages and Systems*, 13(1):99–123, 1991.

[BCC+97] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla. The Ciao Prolog System. Reference Manual. The Ciao System Documentation Series–TR CLIP3/97.1, School of Computer Science, Technical University of Madrid (UPM), August 1997. System and on-line version of the manual available at `http://clip.dia.fi.upm.es/Software/Ciao/`.

[BCHP96] F. Bueno, D. Cabeza, M. Hermenegildo, and G. Puebla. Global Analysis of Standard Prolog Programs. In *European Symposium on Programming*, number 1058 in LNCS, pages 108–124, Sweden, April 1996. Springer-Verlag.

[BDD+97] F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Maluszynski, and G. Puebla. On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*, pages 155–170, Linköping, Sweden, May 1997. U. of Linköping Press.

[BdlBH94a] F. Bueno, M. García de la Banda, and M. Hermenegildo. A Comparative Study of Methods for Automatic Compile-time Parallelization of Logic Programs. In *First International Symposium on Parallel Symbolic Computation*, pages 63–73. World Scientific Publishing Company, September 1994.

[BdlBH94b] F. Bueno, M. García de la Banda, and M. Hermenegildo. Effectiveness of Global Analysis in Strict Independence-Based Automatic Program Parallelization. In *International Symposium on Logic Programming*, pages 320–336. MIT Press, November 1994.

[BdlBH94c] F. Bueno, M. García de la Banda, and M. Hermenegildo. The PLAI Abstract Interpretation System. Technical Report CLIP2/94.0, Computer Science Dept., Technical U. of Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, February 1994.

[BDM97] J. Boye, W. Drabent, and J. Małuszyński. Declarative diagnosis of constraint programs: an assertion-based approach. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*, pages 123–141, Linköping, Sweden, May 1997. U. of Linköping Press.

[BG90] R. Bahgat and S. Gregory. Pandora: Non-deterministic Parallel Logic Programming. In G.Levi and M.Martelli, editors, *Proc. of the 6th International Conference on Logic Programming*. MIT Press, 1990.

[BG00] G. Bolella and J. Gosling. The real-time specification for java. *IEEE Computer*, 33(6), June 2000.

[BHW88] P. Brand, S. Haridi, and D.H.D. Warren. Andorra Prolog—The Language and Application in Distributed Simulation. In *International Conference on Fifth Generation Computer Systems*. Tokyo, November 1988.

[BK96] F. Benoy and A. King. Inferring Argument Size Relationships with CLP(R). In *Proc. 6th International Workshop on Logic Program Synthesis and Transformation*, pages 34–153. Stockholm University/Royal Intitute of Technology, 1996.

[Bla92] Jens Blanck. Abstrakt maskin för Nova Prolog. Internal report, Comp. Sci. Dept., Uppsala Univ., Uppsala, 1992.

[BLM93a] J. Bevemyr, T. Lindgren, and H. Millroth. Exploiting recursion-parallelism in Prolog. In *Proc. PARLE'93*, Berlin, 1993. Springer-Verlag.

[BLM93b] J. Bevemyr, T. Lindgren, and H. Millroth. Reform Prolog: the language and its implementation. In *Proc. 10th Intl. Conf. Logic Programming*, Cambridge, Mass., 1993. MIT Press.

[BM88] Jonas Barklund and Håkan Millroth. Nova Prolog. UPMAL Tech. Rep. 52, Comp. Sci. Dept., Uppsala Univ., Uppsala, 1988.

[Bou00] P. Bouvier. Visual Tools to Debug Prolog IV Programs. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS. Springer-Verlag, September 2000.

[Byr80] L. Byrd. Understanding the Control Flow of Prolog Programs. In S.-A. Tärnlund, editor, *Workshop on Logic Programming*, Debrecen, 1980.

[CA86] W. F. Clocksin and H. Alshawi. A method for efficiently executing horn clause programs using multiple processors, May 1986.

[Car90] M. Carlsson. *Design an Implementation of an OR–Parallel Prolog Engine*. PhD thesis, SICS and the Royal Institute of Technology, S-164 28 Kista, Sweden, March 1990.

[Car93] M. Carro. Implementation of Non-Determinism and Optimization of the Memory Usage in the And-Parallel Execution of Logic Programs. Master's thesis, T. University of Madrid (UPM), Facultad de Informática, Madrid, 28660, December 1993. In Spanish.

[CDD85] J.-H. Chang, A. M. Despain, and D. Degroot. And-Parallelism of Logic Programs Based on Static Data Dependency Analysis. In *Compcon Spring '85*, pages 218–225, February 1985.

[CDO88] M. Carlsson, K. Danhof, and R. Overbeek. A Simplified Approach to the Implementation of And-Parallelism in an Or-Parallel Environment. In *Fifth International Conference and Symposium on Logic Programming*, pages 1565–1577. MIT Press, August 1988.

[CG86] K. Clark and S. Gregory. Parlog: Parallel Programming in Logic. *Journal of the ACM*, 8:1–49, January 1986.

[CG89a] N. Carreiro and D. Gelernter. How to Write Parallel Programs – A Guide to the Perplexed. *ACM Computing Surveys*, September 1989.

[CG89b] N. Carreiro and D. Gelernter. Linda in Context. *Communications of the ACM*, 32(4), 1989.

[CGH92a] M. Carro, L. Gómez, and M. Hermenegildo. Implementation of an Event Driven Scheme for Visualizing Parallel Execution of Logic Programs. In *Primer Congreso Nacional de Programación Declarativa*, pages 262–278, Madrid, Spain, September 1992. FIM/UPM.

[CGH92b] M. Carro, L. Gómez, and M. Hermenegildo. Implementation of an Event Driven Scheme for Visualizing Parallel Execution of Logic Programs. In *JICSLP'92 Workshop on Parallel Execution*, November 1992.

[CGH93] M. Carro, L. Gómez, and M. Hermenegildo. Some Paradigms for Visualizing Parallel Execution of Logic Programs. In *1993 International Conference on Logic Programming*, pages 184–201. MIT Press, June 1993.

[CH83] A. Ciepielewski and S. Haridi. A formal model for or-parallel execution of logic programs. In Mason, editor, *IFIP 83*, 1983.

[CH96] D. Cabeza and M. Hermenegildo. Implementing Distributed Concurrent Constraint Execution in the CIAO System. In *Proc. of the AGP'96 Joint conference on Declarative Programming*, pages 67–78, San Sebastian, Spain, July 1996. U. of the Basque Country. Available from `http://www.clip.dia.fi.upm.es/`.

[CH97] D. Cabeza and M. Hermenegildo. WWW Programming using Computational Logic Systems (and the PiLLoW/Ciao Library). In *Proceedings of the Workshop on Logic Programming and the WWW at WWW6*, San Francisco, CA, April 1997.

[CH98] M. Carro and M. Hermenegildo. Some Design Issues in the Visualization of Constraint Program Execution. In *AGP'98 Joint Conference on Declarative Programming*, pages 71–86, July 1998.

[CH99] M. Carro and M. Hermenegildo. Concurrency in Prolog Using Threads and a Shared Database. In *1999 International Conference on Logic Programming*, pages 320–334. MIT Press, Cambridge, MA, U.S.A., November 1999.

[CH00a] M. Carro and M. Hermenegildo. Tools for Constraint Visualization: The VIFID/TRIFID Tool. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 253–272. Springer-Verlag, September 2000.

[CH00b] M. Carro and M. Hermenegildo. Tools for Search Tree Visualization: The APT Tool. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 237–252. Springer-Verlag, September 2000.

[CLI97] The CLIP Group. Program Assertions. The Ciao System Documentation Series – TR CLIP4/97.1, Facultad de Informática, UPM, August 1997.

[Con83] J. S. Conery. *The And/Or Process Model for Parallel Interpretation of Logic Programs*. PhD thesis, The University of California At Irvine, 1983. Technical Report 204.

[COS96] The COSYTEC Team. *CHIP System Documentation,* April 1996.

[Cro96] J. Crowcroft. *Open Distributed Systems*. University College London Press, 1996.

[CSW88] J. Chassin, J. Syre, and H. Westphal. Implementation of a Parallel Prolog System on a Commercial Multiprocessor. In *Proceedings of Ecai*, pages 278–283, August 1988.

[DA00] P. Deransart and C. Aillaud. Towards a Language for Choice-Tree Visualization and Abstraction. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS. Springer-Verlag, September 2000.

[daC84] Robert daCosta. The history of ada. *Defense Science Magazine*, March 1984.

[DC93] D. Díaz and P. Codognet. A Minimal Extension of the WAM for `clp(fd)`. In *Proceedings of the Tenth International Conference on Logic Programming*, pages 774–790. Budapest, MIT press, June 1993.

[De 89] K. De Bosschere. Multi–Prolog, Another Approach for Parallelizing Prolog. In *Proceedings of Parallel Computing*, pages 443–448. Elsevier, North Holland, 1989.

[DeG84] D. DeGroot. Restricted AND-Parallelism. In *International Conference on Fifth Generation Computer Systems*, pages 471–478. Tokyo, November 1984.

[DeG87] D. DeGroot. Restricted AND-Parallelism and Side-Effects. In *International Symposium on Logic Programming*, pages 80–89. San Francisco, IEEE Computer Society, August 1987.

[DGT96] O. Danvy, R. Glück, and P. Thiemann, editors. *Partial Evaluation*. Number 1110 in LNCS. Springer, February 1996. Dagstuhl Seminar.

[DHM00] P. Deransart, M. Hermenegildo, and J. Maluszynski. *Analysis and Visualization Tools for Constraint Programming*. Number 1870 in LNCS. Springer-Verlag, September 2000.

[DJ94] S. K. Debray and M. Jain. A Simple Program Transformation for Parallelism. In *International Symposium on Logic Programming*, pages 320–336. MIT Press, November 1994.

[DL87] T. Disz and E. Lusk. A Graphical Tool for Observing the Behavior of Parallel Logic Programs. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 46–53. IEEE Computer Society Press, 1987.

[DL90] S.K. Debray and N.-W. Lin. Static estimation of query sizes in horn programs. In *Third International Conference on Database Theory*, December 1990.

[DL93] S.K. Debray and N.W. Lin. Cost analysis of logic programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, November 1993.

[DLGH94] S.K. Debray, P. López-García, and M. Hermenegildo. Non-Failure Analysis for Logic Programs. Technical Report TR CLIP14/94.0, T.U. of Madrid (UPM), Facultad Informática UPM, 28660-Boadilla del Monte, Madrid-Spain, October 1994.

[DLH90] S.K. Debray, N.-W. Lin, and M. Hermenegildo. Task Granularity Analysis in Logic Programs. In *Proc. of the 1990 ACM Conf. on Programming Language Design and Implementation*, pages 174–188. ACM Press, June 1990.

[DN94] M. Ducassé and J. Noyé. Logic programming environments: Dynamic program analysis and debugging. *Journal of Logic Programming*, 19,20:351–384, 1994.

[DN00a] Bart Demoen and Phuong-Lan Nguyen. So many wam variations, so little time. In *Computational Logic 2000*, pages 1240–1254. Springer Verlag, July 2000.

[DN00b] M. Ducassé and J. Noyé. Tracing prolog programs by source instrumentation is efficient enough. *Journal of Logic Programming*, 43:157–172, 2000.

[DNTM89] W. Drabent, S. Nadjm-Tehrani, and J. Maluszynski. Algorithmic debugging with assertions. In H. Abramson and M.H.Rogers, editors, *Meta-programming in Logic Programming*, pages 501–522. MIT Press, 1989.

[Duc91] M. Ducasse. Abstract views of prolog executions in opium. In *Proc. International Logic Programming Symposium*, San Diego, 1991. MIT Press.

[Duc92a] M. Ducassé. OPIUM - an advanced debugging system. In M. J. Comyn, G.; Fuchs, N.E.; Ratcliffe, editor, *Proceedings of the Second International Logic Programming Summer School on Logic Programming in Action (LPSS'92)*, volume 636 of *LNAI*, pages 303–312, Zurich, Switzerland, September 1992. Springer Verlag.

[Duc92b] Mireille Ducassé. A General Query Mechanism Based on Prolog. In M. Bruynooghe and M. Wirsing, editors, *International Symposium on Pro-*

gramming Language Implementation and Logic Programming, PLILP'92, volume 631 of *LNCS*, pages 400–414. Springer-Verlag, 1992.

[EB88] M. Eisenstadt and M. Brayshaw. The Transparent Prolog Machine (TPM): An Execution Model and Graphical Debugger for Logic Programming. *Journal of Logic Programming*, 5(4), 1988.

[EC98] Jesper Eskilson and Mats Carlsson. SICStus MT—A Multithreaded Execution Environment for SICStus Prolog. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Programming Languages: Implementations, Logics, and Programs*, volume 1490 of *Lecture Notes in Computer Science*, pages 36–53, Pisa, September 1998. Springer-Verlag.

[ECR93] ECRC. *Eclipse User's Guide*. European Computer Research Center, 1993.

[EJ99] M. Ducassé E. Jahier. A Generic Approach to Monitor Program Executions. In Danny De Schreye, editor, *International Conference on Logic Programming*, Logic Programming, pages 139–153, Cambridge, Massachussets, December 1999. MIT Press.

[Fab97] Massimo Fabris. CP Debugging Needs. Technical report, ICON s.r.l., April 1997. ESPRIT LTR Project 22352 DiSCiPl deliverable D.WP1.1.M1.1.

[Fag87] B. S. Fagin. *A Parallel Execution Model for Prolog*. PhD thesis, The University of California at Berkeley, November 1987. Technical Report UCB/CSD 87/380.

[FCH96] M. Fernández, M. Carro, and M. Hermenegildo. IDRA (IDeal Resource Allocation): Computing Ideal Speedups in Parallel Logic Programming. In *Proceedings of EuroPar'96*, number 1124 in LNCS, pages 724–734. Springer-Verlag, August 1996.

[Fer94] J.M. Fernández. Declarative debugging for BABEL. Master's thesis, School of Computer Science, Technical University of Madrid, October 1994.

[FHJ91] T. Franzen, S. Haridi, and S. Janson. An overview of the andorra kernel language. In *Extensions of Logic Programming*, pages 163–179. Springer LNCS 596, 1991.

[FIVC98] N. Fonseca, I.C.Dutra, and V.Santos Costa. VisAll: A Universal Tool to Visualise Parallel Execution of Logic Programs. In *Joint International Con-*

ference and Symposium on Logic Programming, pages 100–114. MIT Press, 1998.

[GadlBHM93] M. Garcìa de la Banda, M. Hermenegildo, and K. Marriott. Independence in Constraint Logic Programs. pages 130–146, 1993.

[GC96] G. Gupta and M. Carlsson, editors. *High Performance Implementations of Logic Programming Systems, Special Issue, Journal of Logic Programming*, volume 29. North-Holland, November 1996.

[GH92] G. Gupta and M. Hermenegildo. Recomputation based Implementation of And-Or Parallel Prolog. In *Proc. of the 1992 International Conference on Fifth Generation Computer Systems*, pages 770–782. Institute for New Generation Computer Technology (ICOT), June 1992.

[GH95] L. Gómez-Henríquez. *Sistematización y Uso de las Técnicas de Visualización de Programas Concurrentes*. PhD thesis, School of Computer Science, Technical University of Madrid, 1995.

[GHM93] M. García de la Banda, M. Hermenegildo, and K. Marriott. Independence in Constraint Logic Programs. In *1993 International Logic Programming Symposium*, pages 130–146. MIT Press, Cambridge, MA, October 1993.

[GHPSC94] G. Gupta, M. Hermenegildo, E. Pontelli, and V. Santos-Costa. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *International Conference on Logic Programming*, pages 93–110. MIT Press, June 1994.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability*. W.H. Freeman and Company, 1979.

[GJ89] G. Gupta and B. Jayaraman. Compiled And-Or Parallelism on Shared Memory Multiprocessors. In *1989 North American Conference on Logic Programming*, pages 332–349. MIT Press, October 1989.

[GP95] G. Gupta and E. Pontelli. Data–Parallel Execution of Prolog Programs in ACE. In *IEEE Symposium on Parallel and Distributed Processing*. IEEE Computer Society, 1995.

[GP99] G. Gupta and E. Pontelli. Stack-splitting: A simple technique for implementing or-parallelism and and-parallelism on distributed machines.

In *International Conference on Logic Programming*, pages 290–305. MIT Press, 1999.

[GSC92] G. Gupta and V. Santos-Costa. Cuts and Side-Effects in And-Or Parallel Prolog. *Journal of Logic Programming*, 27(1):45–71, April 1992.

[GSCYH91] G. Gupta, V. Santos-Costa, R. Yang, and M. Hermenegildo. IDIOM: Integrating Dependent And-, Independent And-, and Or-parallelism. In *1991 International Logic Programming Symposium*, pages 152–166. MIT Press, October 1991.

[GSN+88] A. Goto, M. Sato, N. Nakajima, K. Taki, and A. Matsumoto. Overview of the Parallel Inference Machine (PIM). In *International Conference on Fifth Generation Computer Systems*. ICOT, 1988.

[Han77] P. Brinch Hansen. *The Architecture of Concurrent Programs*. 1977.

[Han99] P. Brinch Hansen. Java's Insecure Parallelism. *ACM SIGPLAN Notices*, 34(4):8, April 1999.

[Har96] S. Haridi. A Tutorial of Oz 2.0. Technical report, Swedish Institute of Computer Science, 1996.

[Hau90] B. Hausman. Handling speculative work in or-parallel prolog: Evaluation results. In *North American Conference on Logic Programming*, pages 721–736, Austin, TX, October 1990.

[HB88] Kai Hwang and Fayé Briggs. *Computer Architecture and Parallel Processing*. McGraw-Hill, 1988.

[HBC+99] M. Hermenegildo, F. Bueno, D. Cabeza, M. Carro, M. García de la Banda, P. López-García, and G. Puebla. The CIAO Multi-Dialect Compiler and System: An Experimentation Workbench for Future (C)LP Systems. In *Parallelism and Implementation of Logic and Constraint Logic Programming*, pages 65–85. Nova Science, Commack, NY, USA, April 1999.

[HBC+00] M. Hermenegildo, F. Bueno, D. Cabeza, M. Carro, M. García de la Banda, P. López-García, and G. Puebla. The Ciao Logic Programming Environment. In *International Conference on Computational Logic, CL2000*, July 2000.

[HBPLG99] M. Hermenegildo, F. Bueno, G. Puebla, and P. López-García. Program Analysis, Debugging and Optimization Using the Ciao System Preprocessor. In *1999 International Conference on Logic Programming*, pages 52–66, Cambridge, MA, November 1999. MIT Press.

[HC91] M. Hermenegildo and M. Carro. Experimenting with Independent And-Parallel Prolog using Standard Prolog. In *Jornadas Nacionales de Programación Declarativa*, pages 478–497, Malaga, Spain, October 1991. U. Malaga.

[HC93] M. Hermenegildo and The CLIP Group. Towards CIAO-Prolog – A Parallel Concurrent Constraint System. In *Proc. of the Compulog Net Area Workshop on Parallelism and Implementation Technologies*. FIM/UPM, Madrid, Spain, June 1993.

[HC94] M. Hermenegildo and The CLIP Group. Some Methodological Issues in the Design of CIAO - A Generic, Parallel Concurrent Constraint System. In Evan Tick, editor, *Proc. of the 1994 ICOT/NSF Workshop on Parallel and Concurrent Programming*. U. of Oregon, March 1994.

[HC95] M. Hermenegildo and M. Carro. Relating Data–Parallelism and And–Parallelism in Logic Programs. In *Proceedings of EURO–PAR'95*, number 966 in LNCS, pages 27–42. Springer-Verlag, August 1995.

[HC96] M. Hermenegildo and M. Carro. Relating Data–Parallelism and (And–) Parallelism in Logic Programs. *The Computer Languages Journal*, 22(2/3):143–163, July 1996.

[HCC95] M. Hermenegildo, D. Cabeza, and M. Carro. Using Attributed Variables in the Implementation of Concurrent and Parallel Logic Programming Systems. In *Proc. of the Twelfth International Conference on Logic Programming*, pages 631–645. MIT Press, June 1995.

[HE91] M. T. Heath and J. A. Etheridge. Visualizing the Performance of Parallel Programs. *IEEE Software*, pages 29–39, September 1991.

[Her86a] M. Hermenegildo. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel*. PhD thesis, Dept. of Electrical and Computer Engineering (Dept. of Computer Science TR-86-20), University of Texas at Austin, Austin, Texas 78712, August 1986.

[Her86b] M. Hermenegildo. An Abstract Machine for Restricted AND-parallel Execution of Logic Programs. In *Third International Conference on Logic Programming*, number 225 in Lecture Notes in Computer Science, pages 25–40. Imperial College, Springer-Verlag, July 1986.

[Her87] M. Hermenegildo. Relating Goal Scheduling, Precedence, and Memory Management in AND-Parallel Execution of Logic Programs. In *Fourth International Conference on Logic Programming*, pages 556–575. University of Melbourne, MIT Press, May 1987.

[Her00] M. Hermenegildo. Parallelizing Irregular and Pointer-Based Computations Automatically: Perspectives from Logic and Constraint Programming. *Parallel Computing*, 26(13–14), 2000.

[HF00] S. Haridi and N. Franzén. *The Oz Tutorial*. DFKI, February 2000. Available from `http://www.mozart-oz.org`. Still incomplete at the time of writing this reference.

[HG90] M. Hermenegildo and K. Greene. &-Prolog and its Performance: Exploiting Independent And-Parallelism. In *1990 International Conference on Logic Programming*, pages 253–268. MIT Press, June 1990.

[HG91] M. Hermenegildo and K. Greene. The &-Prolog System: Exploiting Independent And-Parallelism. *New Generation Computing*, 9(3,4):233–257, 1991.

[HN86] M. Hermenegildo and R. I. Nasr. Efficient Management of Backtracking in AND-parallelism. In *Third International Conference on Logic Programming*, number 225 in LNCS, pages 40–55. Imperial College, Springer-Verlag, July 1986.

[HN90] M. Hermenegildo and R. I. Nasr. A Tool for Visualizing Independent And-parallelism in Logic Programs. Technical Report CLIP1/90.0, T.U. of Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, 1990. Presented at the NACLP-90 Workshop on Parallel Logic Programming, Austin, TX.

[HQ91] Philip J. Hatcher and Michael J. Quinn. *Data-parallel Programming on MIMD Computers*. MIT Press, Cambridge, Mass., 1991.

[HR89] M. Hermenegildo and F. Rossi. On the Correctness and Efficiency of Independent And-Parallelism in Logic Programs. In *1989 North American Conference on Logic Programming*, pages 369–390. MIT Press, October 1989.

[HR95] M. Hermenegildo and F. Rossi. Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions. *Journal of Logic Programming*, 22(1):1–45, 1995.

[Hu61] T.C. Hu. Parallel sequencing and assembly line problems. *Operating Research*, 9(6):841–848, November 1961.

[Jan94] Sverker Janson. *AKL. A Multiparadigm Programming Language*. PhD thesis, Uppsala University, 1994.

[JH91] S. Janson and S. Haridi. Programming Paradigms of the Andorra Kernel Language. In *1991 International Logic Programming Symposium*, pages 167–183. MIT Press, 1991.

[JM94] J. Jaffar and M.J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[KA98] D. Krieger and R. Adler. The emergence of distributed component platforms. *IEEE Computer Magazine*, pages 43–53, March 1998.

[Kac84] Peter Kacsuk. A highly parallel prolog based on the generalised DF model. In *2nd Int. Logic Programming Conference*, pages 195–205, Uppsala, Sweden, July 1984.

[Kac90] Péter Kacsuk. *Execution Models of Prolog for Parallel Computers*. Pitman, London, 1990.

[Kac92] P. Kacsuk. Distributed data driven prolog abstract machine (3DPAM). In P. Kacsuk and M.J. Wise, editors, *Implementations of Distributed Prolog*, pages 89–118. Wiley & Sons, Aachen, 1992.

[Kac93] P. Kacsuk. LOGFLOW-2: A transputer based data driven parallel prolog machine. In *Transputer World Congress*, pages 1154–1169, Aachen, 1993.

[Kac94] P. Kacsuk. Wavefront scheduling in LOGFLOW. In *2nd EUROMICRO Workshop on Parallel and Distributed Processing*, pages 503–510, Malaga, 1994.

[Kac97] P. Kacsuk. Granularity control in the LOGFLOW parallel prolog system. In L. Grandinetti, J. Kowalik, and M. Vajtersic, editors, *Advances in High-Performance Computing*, NATO ASI Series, pages 201–218. Kluwer Academic Publishers, Aachen, 1997.

[Kah96] K. Kahn. Drawing on Napkins, Video-game Animation, and Other ways to program Computers. *Communications of the ACM*, 39(8):49–59, August 1996.

[Kal87a] L. V. Kalé. Completeness and Full Parallelism of Parallel Logic Programming Schemes. In *Fourth IEEE Symposium on Logic Programming*, pages 125–133. IEEE, 1987.

[Kal87b] L. V. Kalé. Parallel Execution of Logic Programs: the REDUCE-OR Process Model. In *Fourth International Conference on Logic Programming*, pages 616–632. Melbourne, Australia, MIT Press, May 1987.

[Kar92] R. Karlsson. *A High Performance OR-Parallel Prolog System*. PhD thesis, SICS and the Royal Institute of Technology, S-164 28 Kista, Sweden, March 1992.

[KK84] V. Kumar and L.N. Kanal. Parallel branch-and-bound formulations for and/or tree search. *IEEE transactions on pattern analysis and machine intelligence*, 6:768–778, November 1984.

[Kow96] R. A. Kowalski. Logic Programming with Integrity Constraints. In *Proceedings of JELIA*, pages 301–302, 1996.

[KP96] A. Kusalik and S. Prestwich. Visualizing Parallel Logic Program Execution for Performance Tuning. In Michael J. Maher, editor, *Joint International Conference and Symposium on Logic Programming*, pages 498–512. MIT Press, 1996.

[KS90] A. King and P. Soper. Granularity analysis of concurrent logic programs. In *The Fifth International Symposium on Computer and Information Sciences*, Nevsehir, Cappadocia, Turkey, October (1990).

[KZP97] P. Kacsuk, Zs.Nmeth, and Zs. Pusks. Tools for mapping, load balancing and monitoring in the logflow parallel prolog project. *Parallel Computing Journal*, 22(13), February 1997.

[LC97] A. López and M. Carro. A User Guide to APT. Technical Report CLIP6/97.1, Facultad de Informática, UPM, September 1997.

[LGH95] P. López-García and M. Hermenegildo. Efficient Term Size Computation for Granularity Control. In *International Conference on Logic Programming*, pages 647–661, Cambridge, MA, June 1995. MIT Press, Cambridge, MA.

[LGHD94] P. López-García, M. Hermenegildo, and S.K. Debray. Towards Granularity Based Control of Parallelism in Logic Programs. In Hoon Hong, editor, *Proc. of First International Symposium on Parallel Symbolic Computation, PASCO'94*, pages 133–144. World Scientific, September 1994.

[LGHD96] P. López-García, M. Hermenegildo, and S.K. Debray. A Methodology for Granularity Based Control of Parallelism in Logic Programs. *Journal of Symbolic Computation, Special Issue on Parallel Symbolic Computation*, 22:715–734, 1996.

[Lin97] Z. Lin. Parallel Execution of Logic Programs by Load Sharing. *Journal of Logic Programming*, 30(1):25–51, January 1997.

[LK88] Y. J. Lin and V. Kumar. AND-Parallel Execution of Logic Programs on a Shared Memory Multiprocessor: A Summary of Results. In *Fifth International Conference and Symposium on Logic Programming*, pages 1123–1141. MIT Press, August 1988.

[LL74] J.W. Liu and C. L. Liu. Bounds on scheduling algorithms for the heterogeneous computing systems. In *1974 Proceedings IFIP Congress*, pages 349–353, 1974.

[LMSH97] E. Lamma, P. Mello, C. Stefanelli, and P. Van Hentenryck. Improving Distributed Unification through Type Analysis. In *Proceedings of Euro-Par 1997*, volume 1300 of *LNCS*, pages 1181–1190. Springer-Verlag, 1997.

[LO87] Timothy G. Lindholm and Richard A. O'Keefe. Efficient Implementation of a Defensible Semantics for Dynamic Prolog Code. In Jean-Louis Lassez, editor, *Logic Programming: Proceedings of the Fourth International Conference and Symposium*, pages 21–39. The MIT Press, 1987.

[Lue97] A. López Luengo. Apt: Implementing a graphical visualizer of the execution of logic programs. Master's thesis, Technical University of Madrid, School of Computer Science, E-28660, Boadilla del Monte, Madrid, Spain, October 1997.

[Lus88] E. Lusk et al. The Aurora Or-Parallel Prolog System. In *International Conference on Fifth Generation Computer Systems*. Tokyo, November 1988.

[Lus90] E. Lusk et al. The Aurora Or-Parallel Prolog System. *New Generation Computing*, 7(2,3), 1990.

[Mar00] I. Martín. Una Librería Gráfica para Prolog. Master's thesis, Technical University of Madrid, School of Computer Science, E-28660, Boadilla del Monte, Madrid, Spain, July 2000. In Spanish.

[Mau00] P.M. Maurer. Components: What if the gave a revolution and nobody came? *IEEE Computer*, pages 28–34, June 2000.

[MBdlBH99] K. Muthukumar, F. Bueno, M. García de la Banda, and M. Hermenegildo. Automatic Compile-time Parallelization of Logic Programs for Restricted, Goal-level, Independent And-parallelism. *Journal of Logic Programming*, 38(2):165–218, February 1999.

[MC69] R.R. Muntz and E.G. Coffman. Optimal preemptive scheduling on two processor systems. *IEEE Transactions on Computers*, pages 1014–1020, November 1969.

[Mei96] M. Meier. Grace User Manual, 1996. Available at `http://www.ecrc.de/eclipse/html/grace/grace.html`.

[Mil90] Håkan Millroth. *Reforming Compilation of Logic Programs*. PhD thesis, Comp. Sci. Dept., Uppsala Univ., Uppsala, 1990.

[Mil91] Håkan Millroth. Reforming compilation of logic programs. In V. Saraswat and K. Ueda, editors, *Logic Programming, Proceedings of the 1991 International Symposium*, pages 485–502, San Diego, USA, 1991. The MIT Press.

[MR90] Michael Metcalf and John Reid. *Fortran 90 Explained.* Oxford Univ. Press, Oxford, 1990.

[MR91] U. Montanari and F. Rossi. True-concurrency in Concurrent Constraint Programming. In V. Saraswat and K. Ueda, editors, *Proceedings of the 1991 International Symposium on Logic Programming*, pages 694–716, San Diego, USA, 1991. The MIT Press.

[Nai88] L. Naish. Parallelizing NU-Prolog. In *Fifth International Conference and Symposium on Logic Programming*, pages 1546–1564. University of Washington, MIT Press, August 1988.

[NKD97] Eric Neufeld, Anthony Kusalik, and Michael Dobrohoczki. Visual metaphors for understanding logic program execution. In Wayne Davis, Marilyn Mantei, and Victor Klassen, editors, *Graphics Interface*, pages 114–120, May 1997.

[NT88] Martin Nilsson and Hidehiko Tanaka. A Flat GHC implementation for supercomputers. In R. A. Kowalski and K. A. Bowen, editors, *Proc. Fifth Intl. Conf. Symp. on Logic Programming*, pages 1337–1350, Cambridge, Mass., 1988. MIT Press.

[Ous94] John K. Ousterhout. *Tcl and the Tk Toolkit*. Professional Computing Series. Addison-Wesley, 1994.

[PG94] E. Pontelli and G. Gupta. Nested Parallel Call Optimization. Technical report, New Mexico State University, 1994.

[PG95a] E. Pontelli and G. Gupta. An Overview of the ACE Project. In *Proc. of Compulog ParImp Workshop*, 1995.

[PG95b] E. Pontelli and G. Gupta. On the Duality Between And-parallelism and Or-parallelism. In *Proc. of Euro-Par'95*. Springer Verlag, 1995.

[PG97] A. Pontelli and G. Gupta. Implementation Mechanisms for Dependent And-Parallelism. In Lee Naish, editor, *International Conference on Logic Programming*, pages 123–137. MIT Press, 1997.

[PGH95] E. Pontelli, G. Gupta, and M. Hermenegildo. &ACE: A High-Performance Parallel Prolog System. In *International Parallel Processing Symposium*, pages 564–572. IEEE Computer Society Technical Committee on Parallel Processing, IEEE Computer Society, April 1995.

[PGT+96] E. Pontelli, G. Gupta, D. Tang, M. Carro, and M. Hermenegildo. Improving the Efficiency of Nondeterministic And–parallel Systems. *The Computer Languages Journal*, 22(2/3):115–142, July 1996.

[PH95] G. Puebla and M. Hermenegildo. Implementation of Multiple Specialization in Logic Programs. In *Proc. ACM SIGPLAN Symposium on Par-*

tial Evaluation and Semantics Based Program Manipulation, pages 77–87. ACM Press, June 1995.

[PH99] A. Pineda and M. Hermenegildo. O'ciao: An Object Oriented Programming Model for (Ciao) Prolog. Technical Report CLIP 5/99.0, Facultad de Informática, UPM, July 1999.

[PK96] S. Prestwich and A. Kusalik. Programmer–Oriented Parallel Performance Visualizatoin. Technical Report TR–96–01, CS Dept., University of Saskatchewan, 1996.

[Pon97] E. Pontelli. *High Performance Logic Programming*. PhD thesis, New Mexico State University, 1997.

[Pre94] S. Prestwich. On Parallelisation Strategies for Logic Programs. In Springer-Verlag, editor, *Proceedings of the International Conference on Parallel Processing*, number 854 in Lecture Notes in Computer Science, pages 289–300, 1994.

[PRO] The PROLOG IV Team. *PROLOG IV Manual*.

[Pro98] PrologIA. Visual tools for debugging of Prolog IV programs. Technical Report D.WP3.5.M2.2, ESPRIT LTR Project 22352 DiSCiPl, October 1998.

[Ram98a] J.M. Ramos. VIFID: Variable Visualization for Constraint Domains. Master's thesis, Technical University of Madrid, School of Computer Science, E-28660, Boadilla del Monte, Madrid, Spain, September 1998.

[Ram98b] J.M. Ramos. vifid: Variable Visualization for Finite Domains. Technical Report CLIP4/98.0, Technical University of Madrid, E-28660, Boadilla del Monte, Madrid, Spain, September 1998.

[RC98] J.M. Ramos and M. Carro. VIFID User's Manual. Technical Report CLIP3/98.0, Technical University of Madrid, E-28660, Boadilla del Monte, Madrid, Spain, September 1998.

[RK89] B. Ramkumar and L. V. Kale. Compiled Execution of the Reduce-OR Process Model on Multiprocessors. In *1989 North American Conference on Logic Programming*, pages 313–331. MIT Press, October 1989.

[RN95] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Englewood Cliffs, New Jersey, 1995.

[Rog97] Dale Rogerson. *Inside Com*. Microsoft Press International, January 1997.

[SA00] H. Simonis and A. Aggoun. Search Tree Visualization. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS. Springer-Verlag, September 2000.

[SABB00] H. Simonis, A. Aggoun, N. Beldiceanu, and E. Bourreau. Complex Constraint Abstraction: Global Constraint Visualization. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS. Springer-Verlag, September 2000.

[Sah90] D. Sahlin. The mixtus approach to the automatic evaluation of full prolog. In *Proceedings of the North American Conference on Logic Programming*, pages 377–398. MIT Press, October 1990.

[Sar90] V.A. Saraswat. Concurrent constraint programming. In *Proc. POPL'90*, pages 232–245, 1990.

[Sch97] Christian Schulte. Oz explorer: A visual constraint programming tool. In Lee Naish, editor, *ICLP'97*. MIT Press, July 1997.

[SCH99] G. Smedbäck, M. Carro, and M. Hermenegildo. Interfacing Prolog and VRML and its Application to Constraint Visualization. In *The Practical Application of Constraint Technologies and Logic programming*, pages 453–471. The Practical Application Company, April 1999.

[SCWY91a] V. Santos-Costa, D.H.D. Warren, and R. Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-parallelism. In *Proceedings of the 3rd. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 83–93. ACM, April 1991. SIGPLAN Notices vol 26(7), July 1991.

[SCWY91b] V. Santos-Costa, D.H.D. Warren, and R. Yang. The Andorra-I Engine: A Parallel Implementation of the Basic Andorra Model. In *1991 International Conference on Logic Programming*. MIT Press, June 1991.

[SCWY91c] V. Santos-Costa, D.H.D. Warren, and R. Yang. The Andorra-I Preprocessor: Supporting Full Prolog on the Basic Andorra Model. In *1991 International Conference on Logic Programming*, pages 443–456. MIT Press, June 1991.

[Seq87] Sequent Computer Systems, Inc. *Sequent Guide to Parallel Programming*, 1987.

[SH91] K. Shen and M. Hermenegildo. A Simulation Study of Or- and Independent And-parallelism. In *International Logic Programming Symposium*, pages 135–151. MIT Press, October 1991.

[SH94] K. Shen and M. Hermenegildo. Divided We Stand: Parallel Distributed Stack Memory Management. In E. Tick and G. Succi, editors, *Implementations of Logic Programming Systems*, pages 185–203. Kluwer Academic Publishers, 1994.

[SH96] K. Shen and M. Hermenegildo. Flexible Scheduling for Non-Deterministic, And-parallel Execution of Logic Programs. In *Proceedings of EuroPar'96*, number 1124 in LNCS, pages 635–640. Springer-Verlag, August 1996.

[Sha83] E. Y. Shapiro. A Subset of Concurrent Prolog and Its Interpreter. Technical Report TR-003, ICOT, 1-4-28 Mita, Minato-ku Tokyo 108, Japan, January 1983.

[Sha86] E. Shapiro. Concurrent prolog: a progress report. internal report CS-86-10, Weizmann Institute, April 1986.

[Sha89] E.Y. Shapiro. The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys*, 21(3):412–510, September 1989.

[She92a] K. Shen. Exploiting Dependent And-Parallelism in Prolog: The Dynamic, Dependent And-Parallel Scheme. In *Proc. Joint Int'l. Conf. and Symp. on Logic Prog.*, pages 717–731. MIT Press, 1992.

[She92b] K. Shen. *Studies in And/Or Parallelism in Prolog*. PhD thesis, U. of Cambridge, 1992.

[She96] K. Shen. Overview of DASWAM: Exploitation of Dependent And-parallelism. *Journal of Logic Programming*, 29(1–3):245–293, November 1996.

[SK92] D.C. Sehr and L.V. Kalé. Estimating the Inherent Parallelism in Logic Programs. In *Proceedings of the Fifth Generation Computer Systems*, pages 783–790. Tokio, ICOT, June 1992.

[Smo94] G. Smolka. The Definition of Kernel Oz. DFKI Oz documentation series, German Research Center for Artificial Intelligence (DFKI), November 1994.

[Smo95] Gert Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Computer Science Today*, Lecture Notes in Computer Science, vol. 1000, pages 324–343. Springer-Verlag, Berlin, 1995.

[SMS96] Péter Szeredi, Katalin Molnár, and Rob Scott. Serving Multiple HTML Clients from a Prolog Application. In *Proceedings of the 1st Workshop on Logic Programming Tools for INTERNET Applications*, JICSLP"96, Bonn, September 1996. Available from `http://clement.info.umoncton.ca/~lpnet/lpnet9.html`.

[SS87] L. Sterling and E. Shapiro. *The Art of Prolog*, chapter 3, page 39. Logic Programming Series. MIT Press, fourth edition, 1987. *Numbered as Program 3.4; this number may be different in other reprints*.

[SS90] J. Sundberg and C. Svensson. MUSE TRACE: A Graphic Tracer for OR–Parallel Prolog. Technical Report T90003, SICS, 1990.

[Swe95] Swedish Institute of Computer Science, P.O. Box 1263, S-16313 Spanga, Sweden. *Sicstus Prolog V3.0 User's Manual*, 1995.

[Swe99] Swedish Institute for Computer Science, PO Box 1263, S-164 28 Kista, Sweden. *SICStus Prolog 3.8 User's Manual*, 3.8 edition, October 1999. Available from `http://www.sics.se/sicstus/`.

[Sze89] P. Szeredi. Performance Analysis of the Aurora Or-Parallel Prolog System. In *1989 North American Conference on Logic Programming*. MIT Press, October 1989.

[Tak92] A. Takeuchi. *Parallel Logic Programming*. John Wiley and Sons, 1992.

[Tak00] S. Takashi. Visualizing Constraints in Visualization Rules. In *CP 2000 Workshop on Analysis and Visualization of Constraint Programs and Solvers*, September 2000.

[Tar99] Paul Tarau. Jinni: Intelligent Mobile Agent Programming at the Intersection of Java and Prolog. In *PAAM'9*. The Practical Applications Company, 1999.

[TH87] Tebra and H. Optimistic and-parallelism in prolog. In de Bakker, J. W., Nijman, A. J., Treleaven, and P. C., editors, *Parallel Architectures and Language Europe PARLE*, pages 420–431. Lecture Notes in Computer Science, June 1987.

[Thi86] Thinking Machines Corp., Cambridge, Mass. *The Essential *LISP Manual*, 1986.

[Thi90] Thinking Machines Corp., Cambridge, Mass. *C* Programming Guide*, 1990.

[Tic92] Evan Tick. Visualizing Parallel Logic Programming with VISTA. In *International Conference on Fifth Generation Computer Systems*, pages 934–942. Tokio, ICOT, June 1992.

[Tic95] E. Tick. The deevolution of concurrent programming languages. *Journal of Logic Programming*, 23(2):89–124, May 1995.

[TM99] Tobias Müller. Practical Investigation of Constraints with Graph Views. Poster in International Conference on Logic Programming, December 1999.

[TPG94] D. Tang, E. Pontelli, and G. Gupta. Determinacy Driven Optimizations of Parallel Prolog Implementations. Technical report, New Mexico State University, 1994.

[TPGC94] D. Tang, E. Pontelli, G. Gupta, and M. Carro. Last Parallel Call Optimization and Fast Backtracking in And–parallel Logic Programming Systems. In *ICLP WS on Parallel and Data Parallel Execution of Logic Programs*. Uppsala University, CS Department, Box 311, S–751 Uppsala, Sweden, June 1994.

[TSS87] S. Taylor, S. Safra, and E. Shapiro. A Parallel Implementation of Flat Concurrent Prolog. In E.Y. Shapiro, editor, *Concurrent Prolog: Collected Papers*, pages 575–604, Cambridge MA, 1987. MIT Press.

[UC91] K. Ueda and T. Chikayama. Design of the Kernel Language for the Parallel Inference Machine. *The Computer Journal*, December 1991.

[Ued86] K. Ueda. *Guarded Horn Clauses*. PhD thesis, University of Tokyo, March 1986.

[Ued87] K. Ueda. Guarded Horn Clauses. In E.Y. Shapiro, editor, *Concurrent Prolog: Collected Papers*, pages 140–156. MIT Press, Cambridge MA, 1987.

[VD92] P. Van Roy and A.M. Despain. High-Performace Logic Programming with the Aquarius Prolog Compiler. *IEEE Computer Magazine*, pages 54–68, January 1992.

[Vor92] Andrei Voronkov. Logic programming with bounded quantifiers. In Andrei Voronkov, editor, *Logic Programming—Proc. Second Russian Conf. on Logic Programming*, LNCS 592, Berlin, 1992. Springer-Verlag.

[VPG97] R. Vaupel, E. Pontelli, and G. Gupta. Visualization of And/Or-Parallel Execution of Logic Programs. In *International Conference on Logic Programming*, Logic Programming, pages 271–285. MIT Press, July 1997.

[War80] D.H.D. Warren. An Improved Prolog Implementation Which Optimises Tail Recursion. Research Paper 156, Dept. of Artificial Intelligence, University of Edinburgh, 1980.

[War83] D.H.D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, Artificial Intelligence Center, SRI International, 333 Ravenswood Ave, Menlo Park CA 94025, 1983.

[War87a] D.H.D. Warren. OR-Parallel Execution Models of Prolog. In *Proceedings of TAPSOFT '87*, Lecture Notes in Computer Science. Springer-Verlag, March 1987.

[War87b] D.H.D. Warren. The SRI Model for OR-Parallel Execution of Prolog—Abstract Design and Implementation. In *International Symposium on Logic Programming*, pages 92–102. San Francisco, IEEE Computer Society, August 1987.

[War88] D.H.D. Warren. The Andorra Model. Presented at Gigalips Project workshop. U. of Manchester, March 1988.

[War93] D.H.D. Warren. Logic programming languages, parallel implementations and the andorra model. Invited talk, slides presented at ICLP'93, 1993.

[WH87] R. Warren and M. Hermenegildo. Experimenting with Prolog: An Overview. Technical Report 43, MCC, March 1987.

[Wis86] M. J. Wise. Experimenting with epilog: Some results and preliminary conclusions. In *13th Annual International Symposium on Computer Architecture*, pages 130–139. IEEE Computer Society, June 1986.

[ZTD+92] X. Zhong, E. Tick, S. Duvvuru, L. Hansen, A.V.S. Sastry, and R. Sundararajan. Towards an Efficient Compile-Time Granularity Analysis Algorithm. In *Proc. of the 1992 International Conference on Fifth Generation Computer Systems*, pages 809–816. Institute for New Generation Computer Technology (ICOT), June 1992.