J. LOGIC PROGRAMMING 1994:19, 20:351–384 

# LOGIC PROGRAMMING ENVIRONMENTS: DYNAMIC PROGRAM ANALYSIS AND DEBUGGING

MIREILLE DUCASSÉ AND JACQUES NOYÉ

▷ Programming environments are essential for the acceptance of programming languages. This survey emphasizes that program analysis, both static and dynamic, is the central issue of programming environments. Because their clean semantics makes powerful analysis possible, logic programming languages have an indisputable asset in the long term.

This survey is focused on logic program analysis and debugging. The large number of references provided show that the field, although maybe scattered, is active. A unifying framework is given which separates environment tools into extraction, analysis, and visualization. It facilitates the analysis of existing tools and should give some guidelines to develop new ones.

Achievements in logic programming are listed; some techniques developed for other languages are pointed out, and some trends for further research are drawn. Among the main achievements are algorithmic debugging, tracing for sequential Prolog, and abstract interpretation. The main missing techniques are slicing, test case generation, and program mutation. The perspectives we see are integration, evaluation, and above all, automated static and dynamic analysis. ◁

## 1. INTRODUCTION

Programming environments are essential for the acceptance of programming languages. Programmers, if not provided with the support of proper development and maintenance tools, are more likely to waste time and produce low-quality software. They are there-

*Address correspondence to* Mireille Ducassé, INSA/IRISA, 20 avenue des Buttes de Coësmes, 35043 Rennes Cedex, France or Jacques Noyé, IRISA, Campus de Beaulieu, F- 35042 Rennes Cedex, France. Email: {Mireille.Ducasse, Jacques.Noye}@irisa.fr.

Received May 1993; accepted January 1994.

*THE JOURNAL OF LOGIC PROGRAMMING*

© Elsevier Science Inc., 1994
655 Avenue of the Americas, New York, NY 10010 0743-1066/94/$7.00

fore reluctant to use a language without appropriate programming environments, however powerful the programming language. Logic programming languages are no exception.

Logic programming is continually used as a basis for new languages; people need support to get acquainted with them. Furthermore, and somehow paradoxically, it is the very power of the declarativeness of logic programming which demands powerful support. The discrepancies between the declarative and operational semantics of logic programming languages can be a source of confusion for programmers; tools are required to bridge the gap.

Although logic programming requires programming environments like any other programming language, it offers a better basis than many of them. A clean semantics paves the way for powerful program analysis, which, as discussed in the following, is the core of programming environment tools. Even if not always developed with programming environment considerations in mind, logic program analysis is a flourishing activity. There is undoubtedly a breakthrough potential for programming environments in general.

Research on logic programming environments started relatively early [104] and shows substantial achievements. Original tools are now being adapted to other languages, for example, Algorithmic Program Debugging [168] is being adapted to imperative languages [100] (see Section 6.1). On the other hand, not all the techniques existing for other languages have been adapted to logic programming. The most striking example is a dependency analysis technique used for debugging, called "slicing" [192] (see Section 6.3).

It has been emphasized that, in general, more research on programming environments is needed [148]. This is especially true in logic programming, where research on programming environments has been marginal in comparison to the needs. We believe that programming environments should be one of the top priorities for the logic programming community in the next ten years.

This survey gives a unifying framework which should help to develop further research on the topic. It is subjective and certainly incomplete, but should provide a useful starting point for people interested in logic programming environments. It gives an overview of techniques, tools, and systems which help programmers develop and maintain logic programs. It goes into some details for very basic techniques. Wherever it seems necessary, it mentions techniques developed for other languages, which could be useful for logic programming.

The paper is organized as follows. We first set the scene. In particular, we isolate the three aspects we believe are the core of programming environments: *program development, program analysis*, and *program debugging*. We then concentrate our discussions on program analysis and program debugging. Program analysis is especially emphasized as it is the basis of all the tools. Presentation and visualization of information is also addressed. Lastly, we discuss two general issues: integration mechanisms and evaluation criteria.

## 2. SETTING UP THE SCENE

### 2.1. A User Viewpoint of Programming Environments

The notion of "programming environment" can be understood in many different ways. A common view considers all aspects related to programming, such as language features, compiler, system support, tools, etc., as part of the programming environment. This view is, for example, illustrated by the wide range of topics addressed in a recent workshop about programming environments [102].

Although this view has some foundations, it is too broad to efficiently gather research. We would like to cover a more focused area. In particular, we want to distinguish research

on programming environments from research on languages.

Research on programming environments can contribute significantly to the design of new languages, mainly by pointing out the weaknesses of existing programming languages. A good example is contextual logic programming, which is the main outcome of the Esprit project ALPES on logic programming environments [3]. Contextual programming neatly integrates modules inside Prolog [134]. The types, modules, and meta-programming facilities provided by the declarative language Gödel also owe a lot to programming environment concerns [90]. Another example is λProlog [139] which, among other features, provides "for free" the programming cliché mechanisms described in [7].

There are, however, problems which are not due to weaknesses of programming languages, but to intrinsic weaknesses of the human beings who specify, develop, and maintain programs. We consider that excising language weaknesses is part of language research, and that research on environments should concentrate on *tools to accommodate user weaknesses*. Thus, logic programming environments would still be needed even if the "perfect" logic programming language could be defined.

In particular, *development, analysis*, and *debugging* will always have to be supported. Firstly, some support is needed to help users develop their programs because this is a creative process. There is very little chance that we can ever fully model it. Secondly, automated analysis is needed because, however abstract the programming paradigm, the size of the programs will always end up too large for the human mind. Programmers will always have difficulties in understanding large programs [36]. Lastly, debugging and testing support is needed because, however careful programmers are, they can always make mistakes, either at the specification or programming level.

The classification into development, analysis, and debugging is mainly a guideline to present the survey. The boundaries among the three classes are not clear-cut; for example, development and debugging are intertwined, and debugging requires program analysis. In the following, we will concentrate on program analysis and debugging.

### 2.2. User-Oriented Program Analysis

Program analysis is often performed in the context of compilation, aiming at optimization. When performed in the context of programming environments, the main difference is that the derived properties are ultimately analyzed by human beings. This induces strong requirements about flexibility, response time, and ergonomy. It therefore influences the technical design of the related tools.

User-oriented program analysis is mainly aimed at helping programmers understand programs and their behaviors. They are usually referred to as debugging tools. They, of course, do help programmers to debug because one of the main problems while debugging is understanding programs. However, on the one hand, debugging is more than just understanding, and on the other hand, understanding can be an aim per se. For example, a mode analysis can help with the understanding of the dataflow of a program; a visualization tool can help with the understanding of the operational semantics of a language; when a new feature has to be added in an existing program, the programmer has to understand this program, at least partially, to avoid introducing inconsistencies.

Figure 1 gives a model of user-oriented program analysis. This model is used as a guideline in the following sections, which describe each aspect in some depth. Let us just give a brief overview here. Program analysis uses three sources of data: program source codes, program executions, and program specifications. These data should be available to the analysis component as *object* data. This includes the transformation of specifications

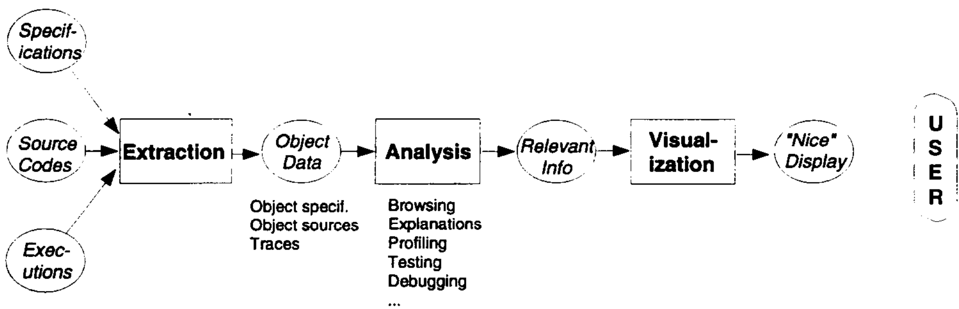


**FIGURE 1.** A model of user-oriented program analysis.

and source codes into an appropriate abstract syntax, as well as the extraction of execution traces. The analysis component can perform browsing, explanation, profiling, testing, debugging, etc. It produces information, relevant to the user at a certain moment, which is displayed in an ergonomical way by a visualization component. This material is eventually examined by the user.

The model emphasizes two important aspects. Firstly, program analysis is not restricted to static analysis, i.e., analysis of source code only; Section 3 stresses that specifications and executions are also interesting sources of data. Secondly, the central component of a programming environment is the *analysis component.*

In our view, the most important objective of a programming environment tool is to *automate as much as possible of the related development and maintenance task* in order to reduce the burden of the programmers. These tasks are usually too difficult to be fully automated. The visualization module is therefore needed to report to the user, but it is not the main part of the environment. We, as users, prefer to examine three lines of *relevant* information rather than a screen full of *nice*, graphically displayed, irrelevant information. Hence, the analysis component should be the main focus of the research on programming environments.

Very often, existing tools are regarded as too low-level or even useless because their analysis component is, actually, too weak. For example, tracers provide a limited number of filtering functionalities which are a degenerate and "hacked" form of analysis. Visualization tools usually provide more sophisticated analysis mechanisms to enable several levels of abstraction, but these mechanisms are still hidden, ad hoc, and intertwined with the graphical mechanisms. With a proper analysis component, the generality of these tools could be greatly improved.

The model is simple, if not simplistic. In particular, users also input requests to the tools. It should be stressed, however, that existing systems are only a degenerate form of this model. No current system satisfactorily covers extraction, analysis, and visualization. We hope that in the near future we will see systems covering and integrating all aspects.

Note that program analysis as considered here is essentially a meta-programming activity; it handles program as data, as well as data as programs. This suggests considering the analysis tools (if not the whole environment) as a meta logic program and the target logic language as the implementation language. Some important benefits can be expected from this approach. In particular, the development of new tools is facilitated by the self-applicability

of the environment and the possibility to reuse software from the object level. This also provides a demanding testbed for the meta-programming facilities of logic programming languages.

## 3. PROGRAM ANALYSIS DATA

As already mentioned, program analysis can use three sources of data: *program source codes*, *program executions*, and *program specifications*. Program source code is a natural candidate; there are indeed many projects on static analysis. In this section, we want to clarify the role of program executions and program specifications.

### 3.1. Program Executions

Program analysis seems often restricted to static analysis, i.e., analysis of program source code, only. Although static analysis has advantages, it is not always the most appropriate type of analysis.

As shown by the popularity of prototyping, it seems, for example, that people often have a better understanding of what a program should *do* rather than of what it should *be*. As understanding is a central issue in user-oriented program analysis, executions which show what programs actually do are an essential source of data.

As a matter of fact, there exist a number of dynamic analysis tools, for example, debuggers, profilers, monitors, or explanation generators. These tools are often ad hoc, with limited scope. They are, however, driven by user demand, and their usefulness is not to be questioned.

An execution is an instantiation of a program with specific input; only some of the potential paths of a program are covered. While this reduces the generality of the analysis, it also reduces its complexity. In some cases, such as bug diagnosis, there is no need to investigate all potential execution paths. Only the paths actually used to produce the error symptoms are relevant. Hence, analyzing executions is sometimes more relevant and more efficient.

This is not to say that dynamic analysis is better than static analysis. Sometimes one is more appropriate, sometimes it is the other one. Our experience shows that, most of the time, a combination of both is actually the most appropriate.

### 3.2. Specifications

The third source of data for program analysis is program specifications, where a program specification is, basically, a high-level description of the *intention* of the programmer.

A big asset of logic programming is that it provides a smooth transition between specifications and programs. In many cases, specifications can be made formal and executable [82], i.e., the distinction between specifications and programs is mainly a question of efficiency and level of abstraction. As a matter of fact, Prolog is sometimes considered as a specification language (see, for example, [13, 129, 93]). Even when, for the sake of expressiveness, specifications cannot be reduced to logic programs, a uniform use of logic provides a path between specifications, (pure) logic programs, and (efficient) Prolog programs. This has been applied with success to program synthesis (see the survey on program synthesis, by Deville and Lau, in this issue [49]).

The link between specifications and programs is currently tightened through a growing understanding of the importance of types [153], which are essential to capture the notion of *meaningless* or *erroneous* data, as well as to reveal meaningful approximations of the relations to be computed. This has led to the introduction of typed versions of Prolog, e.g., Typed Prolog [110], as well as languages which further depart from Prolog, e.g., Gödel [90] or λProlog [139]. The type systems offered by these languages may not, however, be expressive enough to capture the intention of the programmer. Naish suggests, for instance, to include distinguished predicates (called *type predicates*) at the specification level. These predicates are not part of the program level, but can be used to make more precise the intended semantics of the program in different phases of its development [142].

Introducing specifications replaces a big step (programmer's intention to program) by two smaller steps (intention to specification and specification to program), hopefully easier to manage. Capturing the programmer's intention in complete and sound specifications remains a difficult task. In the best case, methodologies are used to enforce rigorous development based on specifications; in the worst case, whether specifications are produced or not depends entirely on the good will of programmers.

As a result, program analysis cannot rely on the availability of *complete* specifications. Nevertheless, there are always some aspects of a system which can be easily specified. Thus, expecting *incomplete* specifications is reasonable and may be beneficial. For example, assertions partially specifying the behavior of a program help reduce user's mediation during algorithmic debugging [56].

## 4. EXTRACTION

Program specifications, program source codes, and program executions cannot be straightforwardly used as analysis data. The *extraction component* transforms them into object data. The role of the object syntax is both to make sure that there will be no confusion between the object program (the program being analyzed) and the meta-program (the analyzer), and to retain information which may be pertinent to the user (e.g., variable print names).

The transformations of program sources and specifications into object data are simple syntactic transformations. Information about program executions is not as straightforwardly available. Program executions are modeled by sets of events, usually called *traces*, or *event histories*. Instrumentation depending on the chosen execution model is required to extract these traces.

In the remainder of the section, we briefly describe each of these mechanisms. They are basic mechanisms for general static and dynamic analysis, essential to programming environments.

### *4.1. Object Data*

The basic role of the transformation into object data is to avoid any confusion between the object-level and the meta-level. It is essentially a naming problem, well known in static analysis and, more generally, in meta-programming [124, 8, 121, 196].

4.1.1. NONGROUND AND GROUND REPRESENTATIONS. The major issue is the representation of object-level variables. The general alternative is between a nonground and a ground representation. Object-level variables are represented by meta-level variables in nonground representations and by ground terms in ground representations.

Ideally, a nonground representation should be typed as advocated by Hill and Lloyd [91] in order to prevent unifying real meta-level variables with meta-level variables representing object-level variables.

Using typeless Prolog, a straightforward choice is to use a ground representation, which has the advantage that the meta-level version of the meta-logical predicates can be given a declarative semantics [91].

A basic ground representation of terms is described by Barklund in [8] as follows:

- The name of a constant `X` is the compound term `constant(X)`.
- The name of a compound term whose functor is `F/m` and whose arguments are `A1` to `Am` is `compound(F, [N1,...,Nm])`, where `N1` to `Nm` are names of the arguments `A1` to `Am`, respectively.
- A name of a variable `X` is a term `var(I)` where `I` is any positive integer, such that the same integer `I` is used for every occurrence of `X` in all terms under consideration, and that all distinct variables in these terms are assigned different names.

Many variants are possible. Actually, van Harmelen advocates user definable representations [187]. For our purpose, the name of constants can be the constants themselves. It is also possible to represent compound terms by compound terms with the same principal functor. It is then necessary to be careful not to confuse the representation of a compound term `var(Term)` and the representation of a variable. A solution consists of applying a mechanism reminiscent of the double quote mechanism used to insert quotes in strings and represent the compound term as `var(var(Term))`. Finally, the representation can also be supported at the implementation level, e.g., by replacing wrappers such as `var/1` by special tags [8].

The use of such a representation is classical in static analysis and meta-programming. However, to our knowledge, only interpreter-based tracers, such as Maeda's [127], have extended its use to the representation of events.

4.1.2. UNIQUE (GLOBAL) IDENTIFIERS. Traced events including variables should not lose variable sharing information. If two events involve the same variable, it is necessary for dataflow analysis that the analysis component can recognize it. For instance, in Figure 2, one expects to see the same name for the first argument of `p/1`, line 1, and the argument of `q/1`, line 4. Basically, each new variable created during an execution should be associated with an identifier unique within the scope of the execution subject to analysis. In the terms of the ground representation defined above, all terms under consideration for the definition of `I` are all the terms involved in the events susceptible to analysis. However, since dataflow analysis only makes sense along a given derivation, distinct variables which are not shared between different derivation paths may be assigned the same name. Let us look again at Figure 2. It does not make sense to look at lines 2 and 9, where `_2 = a` and `_2 = b`, respectively, without understanding that they belong to two different derivations (one from line 2 to 7, and one from line 9 to 12). Therefore, the two different instances of `X`, lines 5 and 12, can share the same identifier without creating any ambiguity.

The standard practice is to use the physical address of a variable as its identifier. Unfortunately, the uniqueness of such identifiers cannot be guaranteed; addresses may change during the execution (e.g., due to a globalization in a WAM-based implementation, or a memory copying or compaction due to garbage collection).

```
p(X, Z) :- Z = a, q(X), fail.                q(f(X)).
p(X, Z) :- Z = b, q(X).

?- p(X, Y).

 1      Call   p(_1, _2)
 2        Call   _2 = a
 3        Exit   a = a
 4        Call   q(_1)
 5        Exit   q(f(_3))
 6        Call   fail
 7        Fail   fail
 8      Next   p(_1, _2)
 9        Call   _2 = b
10        Exit   b = b
11        Call   q(_1)
12        Exit   q(f(_3))
13      Exit   p(f(_3), b)
```

**FIGURE 2.** A simple Prolog trace with variable identifiers.

4.1.3. VARIABLE PRINT NAMES. Usually, programmers carefully choose the *print* names of program variables (i.e., their external, source-level, representation), so that they carry some meaning about the application domain. As the result of an analysis is to be presented to users, it is essential that the variable print names are kept in the representation of object variables. The representation of a variable should therefore be composed of both its print name and its unique global identifier. Prolog/KR [144], KCM [146], and ECLIPSE/SEPIA [131] have implemented such a representation (modulo the problem mentioned above).

## *4.2. Basic Execution Models*

Information about executions is not straightforwardly available. Basic models of executions have to be designed so that extraction mechanisms can extract information related to these models. It should be emphasized again that the basic models are not aimed at users, but at the automated analysis module.

An execution is modeled by a set of atomic actions, or *events*, organized as a *trace*, or *event history*. A sequential execution leads then to a single sequence of events, whereas a parallel execution leads to a set of (local) sequences of events connected by dependency links.

Events can potentially be very low level, e.g., the execution of a machine instruction, a store access, or the execution of a communication primitive. The issue is to design a model which is sufficiently detailed so that the analysis does not miss any information, sufficiently high-level so that the analysis does not spend too much time restructuring or recomputing information, and sufficiently easy to extract so that the resulting response times of the tools are reasonable.

For Prolog, a number of basic models exist. A popular one is the "box model," designed by Byrd [26] and described formally by Tobermann and Beckstein [182]. This model shows a procedure execution as a black box (hence its name). Events are related to goals; there are four types of events (traditionally called "ports"). For a given goal $g$, the meaning of the events is as follows:

`Call g:` $g$ is being invoked;
`Exit g:` $g$ has just been proved;
`Redo g:` the execution backtracks to *a subgoal of* $g$;
`Fail g:` $g$ could not be proved.

Another model is closer to the operational semantics of Prolog. It shows backtracking in the order in which it actually appears in the execution. The only difference with the box model is the backtracking port `Next`, which has a slightly different semantics from `Redo`:[1]

`Next g:` the execution backtracks to $g$.

The difference between the two models is subtle but significant. Let us illustrate it with an example. Figure 3 shows a program traced using the five ports previously defined. One can see that for the facts (e.g., `s`) the `Redo` and `Next` events are consecutive and seem redundant. But in large executions, goals with a nonempty body are usually more numerous than facts. If such goals have choice points (e.g., `q`), the two backtracking events related to these goals occur at very different places. Here, re-entering the box of `q` is shown at line 7 (before the very first backtracking to a subgoal of `q`), while the actual backtracking to `q` is shown at line 11. The deeper the subtree to prove goals such as `q`, the larger the distance. Note that if line 11 is missing from the trace (`Next q`), which is the case in the box model, the invocation of `t` on line 12 seems to arrive by magic.

The two ports are useful. The `Next` port is more faithful to the operational semantics of Prolog; it tells where backtracking actually occurs. The `Redo` port is very useful to trace "breadth-first." In particular, if the details of the execution of a goal are hidden, the `Redo` port captures backtracking information which otherwise would have been lost [188]. For automated analysis, both ports should be extracted and clearly separated as in the example.

Many refinements to the previous models can be made. For example, Eisenstadt [69] defines six types of failure: "subgoals of $g$ have failed; system primitive has failed; subgoal has backtracked to cut, failing parent; no definition in database; definition exists but different arity; arity OK but no resolvents found; and clause would have been considered but does not unify." Morishita and Numao [135], as well as Schleiermacher *et al.* [164], add information at the clause level, in particular about unification. The box model has been adapted to functional logic programming by Hanus and Josephs [88].

For some analyses, e.g., algorithmic debugging, the control flow information is of little interest and a declarative model is more appropriate than an operational one. In such a case, the resulting proof tree can be sufficient. Sometimes, it is also important to have information about intermediate proof trees (i.e., before failures) or to see how the resulting proof tree has been constructed. In such cases, one can use a degenerate version of the

[1] Note that the two ports are often given the same name in different models, which is a source of confusion.

```
q :- s.    q :- t.      s.

?- q, r.

 1      Call  q
 2        Call  s
 3        Exit  s
 4      Exit  q
 5      Call  r
 6      Fail  r
 7      Redo  q            % *** re-entering the box of q
 8        Redo  s          % re-entering the box of s
 9        Next  s            % backtracking to s
 10       Fail  s
 11     Next  q              % *** backtracking to q
 12       Call  t
 13       Fail  t
 14     Fail  q
```

**FIGURE 3.** An operational model with two types of backtracking events.

previous operational models and keep only the `Exit` ports.

This list is certainly not exhaustive, and each Prolog tracer actually uses a different variant of the previous models with more or less information. Each model has its advantages, but none of them is absolutely better than all the others. The basic model to be used depends on many parameters, in particular on the type of analysis which is to be performed.

### *4.3. Tracing Mechanisms for Sequential Executions*

In the following, we list and briefly discuss some mechanisms to extract trace information from sequential executions. A more thorough discussion can be found in [67].

4.3.1. Manual Program Source Instrumentation. A straightforward way to extract information about executions of a program is to instrument its source code. The most primitive way is to let users insert "write" statements into their programs. This can be very precise when users know exactly what information is needed and insert write statements at appropriate places, but this manual treatment can become very tedious. It usually requires a lot of trials and errors to find the relevant places.

A more sophisticated way consists of inserting (still by hand) "trace" statements instead of "write" statements. The trace statements can then be programmed to send information to the analysis component. Nevertheless, letting the user insert the trace statements is not appropriate for automated dynamic program analysis. The trace information has to be generated automatically so that the analysis component has some guarantees that the required information will be extracted.

4.3.2. AUTOMATIC PROGRAM SOURCE INSTRUMENTATION. The next step is to automatically instrument programs. A technique to trace according to the box model described in the previous section is to encapsulate examined predicates. For example, the definition of predicate `p/1` can be replaced by

```
p(X) :-
    ( trace(call, p(X)) ; trace(fail, p(X)), fail ),
    p_do(X),
    ( trace(exit, p(X)) ; trace(redo, p(X)), fail ).

p_do(X) :-
    <p body>.
```

The `trace/2` predicate succeeds once. When `p/1` is invoked, `trace(call, p(X))` is invoked and a `Call` line is traced, then `p/1` is executed normally. If `p/1` succeeds, `trace(exit, p(X))` is invoked; otherwise, `trace(fail,p(X))` is invoked and the failure is propagated. If `p/1` succeeds and the execution later backtracks, `trace(redo,p(X))` is first invoked, then `fail/0` forces a backtracking to `p/1` itself.

Note that it is the current substitution of the variables which is traced. This is, in general, what is needed; the `Redo` and `Fail` events, however, need some refinements. This mechanism is also more sophisticated for full Prolog.[2]

Note also that, as mentioned before, if one is only interested in the construction of the proof tree, tracing the `Exit` events is sufficient. The transformed `p/1` predicate is then

```
p(X) :-
    p_do(X),
    trace(exit, p(X)).
```

The `trace/2` predicate is programmed to pass the information to the analysis component. The transformed code can be compiled, and therefore runs reasonably efficiently, although the method leaves choice points at each predicate even when it is determinate, impeding last call optimization. This solution is not too difficult to implement and is satisfactory for occasional breakpoints.

When only the resulting proof tree is of interest, another transformation consists of adding an extra argument to each predicate, which collects the proof tree. The predicates

```
p :- q, r.
q :- ( s1 ; s2 ).
```

are transformed into something like

[2]Interested readers can see the "advice" utility written by R. O'Keefe which is part of the DEC10 Prolog library, available by anonymous ftp from the AIAI of the University of Edinburgh (`aiai.edinburgh.ac.uk`).

```
p(proof(p, (ProofQ, ProofR))) :- q(ProofQ), r(ProofR).
q(proof(q, ProofS)) :- ( s1(ProofS) ; s2(ProofS) ).
```

Note that this can only answer questions of the type "why X" for programs without negation. In order to answer questions of the type "why not X," and to be able to trace programs with negation, failing search trees have to be explicitly stored [190, 172].

As discussed by Specht [172], the latter approach seems to be particularly adapted to tracing bottom-up evaluations such as in deductive databases. Indeed, in bottom-up evaluations, one cannot rely on the execution order to trace information by side effect. Furthermore, in deductive databases, the program is typically heavily transformed before it is compiled. Such a program instrumentation allows the information to be traced in terms of the initial program and not in terms of the transformed one.

4.3.3. INSTRUMENTATION OF META-INTERPRETERS. A natural step is to use the same techniques as before, but within a meta-interpreter. An introduction to meta-interpreters can be found, for example, in [174, chapter 19]. Typically, a meta-interpreter contains some clauses to recursively process lists of goals, and a clause which reduces a single goal. This clause can be instrumented with the same tracing instructions as above.

```
solve(Goal) :-
    ( trace(call, Goal) ; trace(fail, Goal), fail ),
    reduce(Goal),
    ( trace(exit, Goal) ; trace(redo, Goal), fail ).
```

This can be generalized to multilayered meta-interpreters [14, 175, 34].

In the same way, the construction of the proof tree in an argument, mentioned in the previous paragraph, can be done by a meta-interpreter [19, 167]. The `solve/2` predicate carries on a second argument which is constructed inside the `reduce/2` predicate as follows:

```
reduce(Goal, proof(Goal, ProofBody)) :-
    clause(Goal, Body),
    solve(Body, ProofBody).
```

Unfortunately, the space and time optimizations of the compiler are mostly lost. Thus, large programs that can run when compiled may require too much memory when interpreted by a meta-interpreter. If the only tracing means is a meta-interpreter, programmers may be left with programs which they cannot trace. Furthermore, for user-oriented analysis, response times must be kept reasonable; hence, the lack of efficiency is a major problem.

Partial evaluation [103, 84, 161, 173, 124], used on the instrumented meta-interpreter and the program to be traced, can produce a program equivalent to the instrumented program of the previous section. Partial evaluation plus instrumented meta-interpreters is more generic than a specialized program instrumentation. One does not need to write a specialized program transformation, but can benefit from a general partial evaluator. Furthermore, if the simple instrumented meta-interpreter is efficient enough to trace the analyzed program, then the partial evaluation step can be skipped.

Enhancing standard meta-interpreters to get high-level information is very easy. They are therefore useful to prototype particular analyses. For example, [174, chapter 19] presents a meta-interpreter which detects stack overflows, one which diagnoses false solutions bottom-up, one which diagnoses false solutions top-down, and one which diagnoses missing solutions.

4.3.4. COMPILED CODE INSTRUMENTATION. The implementation of the tracer can be pushed further down. If an interpreter implemented in a low-level language is used, this interpreter can be modified to generate trace information. Alternatively, if a compiler is used, the compiled code can be modified for the same purpose. In the case of byte-code compilation, the emulator can also be modified.

Notification points are set into the compiled code when an interesting event is reached. Note that not every interesting event corresponds to a physical location in the compiled code. The most obvious example is the `Exit` port. All reasonable Prolog compilers implement last call optimization via a continuation mechanism; the last subgoal call of a clause does not return to its caller, but the execution directly proceeds with the continuation of the call. Hence, there is no location in the code where the debugger could be notified when the last subgoal of a clause exits.

In the tracer of the ECLiPSe/SEPIA Prolog system [131], the `Call` port corresponds directly to a call instruction, while only facts notify the debugger when they exit. The tracer uses its own stack of `Call` and `Exit` frames to reconstruct the other ports.

Compiled code instrumentation has given, so far, the best results in terms of both efficiency and space consumption. It has also led to more precise tracing. For example, cuts can be traced showing precisely which choice points are actually removed. This solution, however, takes notably more time and effort to implement. The resulting tracer is also highly dependent on the implementation of the interpreter or compiler with which it is tightly connected. It is therefore worth considering the other solutions further, all the more as program transformation and partial evaluation techniques have great potential.

### *4.4. Tracing Mechanisms for Parallel Executions*

Tracing parallel executions exhibits further difficulties due to the "probe effect," nonrepeatability, and the lack of a synchronized global clock [130]. The first two problems can be alleviated by resorting to execution replay. The latter one requires a careful organization of the trace, capturing the partial ordering of the recorded events.

4.4.1. EXECUTION REPLAY. The probe effect and nonrepeatability are actually two aspects of the same problem: parallel executions inherently lead to *races*, i.e., competitions for the access of some shared data, for example, OR-parallel access to a choice point, or AND-parallel access to an unbound variable. The resolution of a race depends on the relative speeds of the competitors, and is therefore sensitive to any factor modifying these relative speeds, e.g., the state of the memory hierarchy or the amount of network traffic.

This problem is especially acute with concurrent logic languages where indeterminism, through the committed choice mechanism, is a basic language feature. If several clauses match a goal, the chosen clause depends on the availability of the goal variable bindings and the relative speeds of the parallel clause matching activities. As a result, running a program twice can lead to different behaviors of the program. The problem is not absent from parallel Prolog systems either. In OR-parallel systems, such as Aurora [32], the relaxation

of the order in which the solutions are produced, combined with the use of cuts or one-solution constructs, also leads to programs returning different results, here different answer substitutions, including failure, from one run to the other. The introduction of asynchronous built-in side effects (I/O, assert/retract) adds some more sources of indeterminism.

Repeatability remains an issue even when there is no indeterminism in the language (or in the program). In that case, the results are repeatable, but the paths followed to get to these results may still be different. Let us consider the AND-parallel execution of two goals for both of which there is no successful derivation. Failure may then be caught alternatively while solving one goal or the other. At an even lower level of observation, the paths may be the same, but the scheduling of work between the processing agents different. This cannot be ignored when tracing is done with performance debugging in mind or when the system itself is being debugged.

Finally, since tracing is both time and space consuming, switching tracing on may also influence race resolution and modify the behavior of the traced program. This is the probe effect. In general, a trace does not reflect the execution without tracing, and two executions of the same program do not result in the same trace.

Control driven execution replay [115, 118] is a generic method to deal with indeterminism in parallel programs. During an *initial execution*, each process involved in the computation records a minimal trace, called *history tape*, collecting the result of all the race resolutions, as well as program input. The race-related events are very simple. In case of a shared memory implementation, each event corresponds to a shared object access. It records the object identifier and its version number. In case of a message-passing implementation, each event corresponds to a message reception. It records the sender identifier and the message number (local to the sender). This simplicity, together with the lack of global bottleneck, makes the process very lightweight, limiting the probe effect. The history tapes can then be used to guide new executions, called *replays*, which are equivalent to the initial execution. In particular, detailed tracing can be safely performed during replay.

Execution replay has been implemented in the parallel Prolog system PEPSys [9], based on Instant Replay [115], the seminal version of execution replay based on shared objects. Execution replay was mainly seen as a tool to help debug the system itself. In particular, it captured scheduling decisions. Interestingly, exploiting specific properties of the PEPSys computational model made it possible to implement a simplification of the initial instant replay scheme [44]. This resulted in a very low overhead for the initial execution, typically 2–3%, ensuring, in general, an identical behavior of the system with and without recording on.

Execution replay has also been considered in the context of concurrent logic languages. In [167], Shapiro describes a simple Flat Concurrent Prolog meta-interpreter which, combined with a source transformation of a (noninteractive) program, computes, as a tree data-structure, a trace recording the order of goal reductions. This makes it possible to reconstruct the computation, e.g., to apply algorithmic debugging techniques. In the same spirit, Gaifman *et al.* give a transformation which captures the indeterminism of committed choices [83]. The transformation is justified by a study of execution replay in an abstract setting, showing in particular that execution replay of concurrent logic/constraint programs is simplified by a monotonicity property of these programs. Preliminary experimental results on a stream merger running on the Concurrent Prolog system Logix [170] give some upper bounds of the cost of the transformation: a 50% overhead on the initial execution and 25% on the replay. Considering the highly nondeterministic nature of the application, the transformation should be reasonably effective in real-life applications including sizable deterministic computations. Let us note that both proposals are weak forms of execution

replay, as previously described. They do not allow for a parallel replay of the execution with an equivalent scheduling.

Execution replay makes it possible to remove indeterminism from parallel tracing, allowing cyclical debugging, while limiting the probe effect. It also makes it possible to relate different kinds of traces of the same program, leading to a better integration of the debugging tools. Leu and Schiper [117] show, for instance, how to integrate visualization and symbolic debugging. Execution replay should be a basic component of any parallel environment.

4.4.2. THE LACK OF GLOBAL TIME. It is essential that the analysis component can move forward and backward from a given event. This is easy when considering sequential executions. Events are usually time stamped with a chronological event number and the trace is organized as a sequence of events. Such an organization is still possible with parallel executions. It requires either a global (possibly logical) clock, easily available on a shared memory machine, or totally ordered logical clocks, as defined by Lamport [111]. An interesting alternative consists of organizing the global trace as a set of local traces connected together through links corresponding to communication between the parallel activities, in the spirit of the history tapes of execution replay. Such an organization is more accurate in that it does not impose an arbitrary total order on the trace, but keeps the causal dependencies between the events. A further step would be to time stamp the events with partially ordered logical clocks [79, 157], or to recreate these time stamps at analysis time (in case of post mortem analysis of a full trace), and take advantage of the applicability of these clocks to concurrency measures and global state analysis.

### *4.5. Connection with the Analysis Component*

The model given in Figure 1 is a conceptual model only. It is very costly to systematically extract and send all the pieces of possibly interesting information regarding specifications, sources, and executions to the analysis module each time something new is occurring. Traces and sources can be very large, and any particular analysis requires, in general, only selected information. Thus, communication between the two components has to be driven by the analysis component.

Partial solutions exist for analysis of executions traces. Ducassé has designed a trace query mechanism which optimizes control flow analysis [61]. Kusalik and Oster provide parameterizations of the extraction components, based on meta-interpreters [107]. Solutions derived for imperative languages in the Dalek system [147] could be useful for logic programming.

The problem, even if less acute, also exists for source access. When program sources get really large, it is inappropriate to parse them each time a property has to be derived. Database systems could be used to store already parsed pieces of code.

## 5. SOME USER-ORIENTED PROGRAM ANALYSES

The following list of analyses focuses on aspects which have not been given much attention so far in the community: browsing, explanations, profiling, and reverse-engineering. It does not pretend to be exhaustive. In particular, we conjecture that many existing static analyses developed within the framework of abstract interpretation (see, for example, [132, 21, 39, 22, 89]) could derive properties interesting for end users.

Our aim is not to discuss how analyses are implemented, but to give an indication of what they can be used for in the context of programming environments. Debugging and testing, which are particularly important for programming environments, are addressed in the next section..

## 5.1. Browsing

For experienced programmers, it is usually easy to gain local understanding of programs. This is especially true in Prolog where variables are local to a clause and where most systems now have some sort of modularity. When the size and number of modules increase, programs become harder and harder to understand. Users need some support to gain a global understanding, e.g., of the relationships between predicates and between modules.

A strong ergonomical requirement is that, at each moment, a user should not have to face more than *seven* items of information (plus or minus two) [133]. The nature of the items depends on the skill of the concerned user.

Browsers help users traverse stepwise both source and executions. Step-by-step tracing, for example, is a very primitive way of browsing. Executions and sources can be abstracted so that, at each browsing step, the user sees a "chunk" of information which gives meaningful information of reasonable size.

The issue is then to define abstraction criteria. For Prolog executions, some necessary abstraction criteria have been identified by Bergantz and Hassell [11]: control flow, data flow, data structure, and function relations. Control flow, data flow, and data abstracts have been implemented in the Opium system [58], together with failure and endless loop analyses which produce symptom-oriented abstracts. Höök *et al.* propose a recursion abstract and a dataflow abstract [95]. Top-down zooming is a common browsing strategy [5, 70, 123]. It starts from the topmost goals and recursively zooms into their executions whenever the analysis requests it. Most graphic systems discussed in the visualization section provide browsing facilities. The transparent Prolog Machine (TPM) has two levels of granularity and provides ad hoc abstraction mechanisms [16]. In a related field, [122] restructures proofs made by theorem provers. For imperative languages, the Traceview system [128] provides a number of abstract points of view.

The usual source browser is a cross referencer as, for example, Xref, written by Dave Bowen *et al.*, and part of the AIAI Prolog library. In our point of view, much more could be done in source browsing than cross referencing. For example, it would be very helpful, when looking at recursive predicates, to automatically abstract away all the arguments which do *not* take part in the recursion.

## 5.2. Explanations

Explanation systems are usually based on high-level browsing, but go further in the direction of users.

Explanations are essential for teaching purposes. Novice programmers have misconceptions about the language they use [189, 94]. This can be overcome by an appropriate model [25], or by tutoring systems [43]. Explanations can also be very helpful for experienced programmers to debug programs [197, 72, 59].

However, explanations are mostly required to help end-users of sophisticated programs to accept the results of computations. The problem is well known for expert systems, but it seems to be identical for other areas, such as deductive databases or systems using

constraints. Unless appropriate explanation tools are provided, the new technologies will not break through.

For deductive databases, the study has started [169, 193, 172, 4], but people are mainly busy extracting the resulting proof tree. No sophisticated explanation system has been built yet.

Logic programming has been used to support explanations for expert systems [42, 73, 163, 175, 196]. Conversely, studies of explanations for expert systems could certainly contribute to the explanations of logic programs; see, for example, [159, 178, 149].

### *5.3. Profiling*

Profiling consists of investigating where the programs consume most of their time or space in order to know, for example, what needs to be optimized by hand. Although clever compilers reduce the need for profiling, there is always a need for tools to understand where computations spend time. For example, a program written in a style as declarative as possible will not, in general, be sufficiently efficient. However, there is no need to optimize all of it (at the risk of making the program too complicated and hard to maintain). Usually, 80% of a computation is spent in 20% of the code. Identifying this 20%, and only optimizing it, is a significant gain in time and maintainability.

Profilers give some insight into the time spent in units of the code [87]. This information can be seen as a sort of trace. It is, however, of a lower level than the information usually provided by tracers. Nevertheless, this information is basic, and should be extracted by the same component as the other trace. Once extracted, it has to be analyzed in the same flexible way by the analysis component. It is a problem that existing profilers require the users to perform most of the analysis by hand.

Most existing logic programming systems provide a profiler which gives some statistics about occurrences of events and measurements of time; see, for example, ParaGraph [2] for a multiprocessor profiler. Most profilers are dynamic only. Gorlick and Kesselman [85] and Tick [181] combine static and dynamic analysis. These systems, however, lack, in general, the flexibility of a generic analysis component. Cohen and Carpenter [40] developed a language to inquire about the run-time behavior of Algol programs. This language allows more general analysis to take place.

In the context of parallel systems, profiling is also useful to estimate the inherent parallelism of an application and even give parallelization hints [177, 75, 165, 108].

### *5.4. Reverse-Engineering*

Prolog has been successfully used for reverse-engineering Cobol programs [17, 31]. Conversely, although the number of logic programs has not yet reached the critical mass which makes reverse-engineering an absolute necessity, one could imagine that reverse-engineering tools are designed for logic programming to help structure or re-structure existing code.

## 6. AUTOMATED DEBUGGING AND TESTING

Debugging and testing are central issues of programming environments. We see testing as the process of analyzing a program with the intent of *detecting* errors, while debugging is the process of analyzing a program to *locate and fix* the detected errors.

In this section, we review strategies and tools which are aimed at automating these costly activities. The boundary between testing and debugging is not easy to define. For example, some strategies are able to detect and locate at the same time. In the following, testing and debugging are therefore usually mixed.

In a survey of automated debugging covering declarative and imperative languages [62], we have identified three main strategies for automated debugging: verification with respect to specification, checking with respect to language knowledge, and filtering with respect to the error symptom. The *verification* strategy compares the actual program with some specification of the intended program. The *checking* strategy looks for suspicious places which do not comply with some explicit knowledge of the programming language. The *filtering* strategy filters out parts of the code which cannot be responsible for the error symptom. To these three strategies we have added a section on automated generation of test data, as executing programs with appropriate test data happens to be the most commonly used testing strategy in practice.

## 6.1. Verification with Respect to Specification

In the context of logic programming, the most active trend in verification is algorithmic program debugging, based on the initial work of Shapiro [168]. It compares the execution of a program against its specification. It assumes that there is such a complete specification. As it is, in general, impossible to have a complete formal specification of a program, it is actually assumed that the user can complement the formal specification and act as an "oracle." Algorithmic debugging, in its simplest form, runs an actual computation; at each computation step, the assertions derived from this computation are presented to the user, who decides whether they hold. If the program is incorrect, and if the user can correctly answer all the questions, then an erroneous component is found. It is the part of the program whose behavior is incorrect, but whose components all exhibit a correct behavior.

Following Shapiro's work, a number of studies have been made for Prolog [76, 123]. Algorithmic debugging is used in tutoring systems [125]. As the number of queries may be rather large, some systems use heuristics to ask users more relevant questions first [151, 28, 27, 92, 152]. Another way to reduce the number of queries is to use partial formal specifications as partial oracles [57, 56].

Algorithmic debugging has been extended to concurrent logic programming [96, 180, 120, 74], concurrent constraint logic programming [81], deductive databases [169], and Gödel [12]. Some preliminary work has been done to automatically fix programs when complete specifications are available [48, 116]. Although this assumption seems a bit unrealistic for real programs, this work could be useful for tutoring systems. Algorithmic debugging has also being adapted to imperative languages [100, 166] and lazy functional languages [141, 145].

Algorithmic debugging is an interesting technique. The current implementations, however, lack some user support. For example, users may make mistakes while acting as oracles. There must therefore be some sort of checking of their answers to validate the diagnosis of the algorithm.

Type and mode checking are other important incarnations of the verification strategy [143, 110, 18, 153].

## 6.2. *Checking with Respect to Language Knowledge*

The checking strategy systematically parses programs and searches for language-dependent errors. The knowledge of some aspects of the language is represented by consistency rules which model how these aspects should be implemented or should behave. These rules capture well-formedness knowledge which could not be encoded in the compiler. Some of the rules may assume that programming conventions are adhered to. This technique checks program sources and program executions against the consistency rules. The parts of the program which do not conform are suspicious. The rules used to detect the errors provide some explanation of the mistake.

The larger the programs, the more likely it is that they will deviate from stereotyped programming styles, even if coding standards are adhered to. Hence, an important issue for this technique is to choose the sensitivity of the checking: how much should a part of a program deviate from a rule to be suspected? If the technique is too sensitive, it will be oversuspicious; if it is not sensitive enough, it will miss many errors.

In general, the suspicious code is not necessarily erroneous. Furthermore, some errors cannot be detected by a system which only uses language knowledge. Last but not least, there is no criterion to evaluate the completeness of the set of checking criteria. Therefore, there is no limit on the number of aspects to be checked. This can result in the bad situation where none of the errors has been detected while most of the program is suspicious with respect to several criteria. In such a case, the programmer would have to analyze a lot of warnings to no avail.

As already mentioned, all the analyses performed by abstract interpretation could be applied here. The most typical examples are type and mode inference; see, for example, [23, 46]. They are very often thought of for optimization purposes, but they can be very useful for debugging. The same applies to optimization analyses for concurrent and parallel languages, e.g., [38].

Stereotyped bug recognition has been studied for Prolog [70, 179, 125, 72]. For concurrent languages, deadlocks have been the main type of problem investigated [97, 98].

## 6.3. *Filtering with Respect to the Error Symptom*

Once a symptom of error has been detected, the filtering strategy assumes the correctness of the program components which cannot have produced the error symptom (or which are unlikely to produce it). This strategy does not suspect any code, but is aimed at reducing the amount of code which has to be examined by users.

The most important filtering technique is *slicing*, which is a symptom-driven dependency analysis [192]. So far, it has been addressing two symptoms: "wrong valued output variable" and "wrong control sequence." Slicing prunes out the parts of the program which the wrong value, respectively the wrong control sequence, does not depend on. What remains from the program is called a slice. Slicing according to a wrong valued variable uses a dataflow analysis, and slicing according to a wrong control sequence uses a control-flow analysis. The slice can be computed symbolically using the program source only, or it can be computed on the particular program behavior which has exhibited the symptom under study. A slice does not always contain the erroneous statement [105], but it is always informative.

At present, there exists no slicing algorithm for Prolog or any other logic programming language. There are enough flow analyses so that this could be done promptly; see, for example, [46]. Slicing algorithms for imperative languages can be found in [192, 155, 6]. Slicing for concurrent and parallel imperative languages has also been studied [68, 35, 51].

Heuristic filtering consists of making *a priori* hypotheses about parts of the program which may not have caused the detected symptom in order to restrict the search space for further investigations. This technique is only acceptable if there is a possibility of backtracking over the hypotheses when further analysis does not succeed in finding the error. Pereira and Calejo have applied some heuristic filtering to logic programs, in complement with algorithmic debugging [150, 28].

### *6.4. Test Cases*

A restrictive definition of testing is given by [138]: *Testing is the process of executing a program with the intent of finding errors.* There are testing strategies other than simply executing the program. Verification and language checking as defined earlier are good means to detect errors. However, even powerful verification and checking tools are incomplete. Executing programs with appropriate test data cannot be avoided. Therefore, support is needed to generate systematic test data.

Test cases should ensure that as many errors as possible are detected. It should integrate several techniques, and should be able to tell users what has been tested so far, and what confidence on which part can be assumed.

Software testing is a flourishing field of software engineering, and logic programming is used as a basis for testing other languages [86, 129, 176]. However, work on testing methodologies applied to logic programming languages is just starting [10].

Testing methodologies and techniques are essential for the acceptance of Prolog and other logic programming programs as "real" programming languages. For example, program mutation [47] should be investigated . Program mutation consists of changing small parts of a program and seeing whether the program still behaves correctly (or incorrectly).

## 7. PRESENTATION AND VISUALIZATION OF INFORMATION

Once program specifications, sources, and executions have been analyzed and properties derived, the results have to be presented to the user. In this section, we present some ergonomical presentation and visualization concepts.

We consider both textual and graphical presentation means. Indeed, some people have poor abilities to decipher 3D diagrams. It has been shown that these people are penalized when browsing through hypertext systems [30]. Therefore, environments should provide both textual and graphical tools to accommodate all kinds of users.

In the following, we distinguish two types of visualization: application independent and application dependent. We believe that the issue is really to provide the support for application-dependent visualization. For example, 80% of the code written for CHIP [52] applications is typically dedicated to visualization.

### *7.1. Application-Independent Visualization*

We review in this section the general visualization tools.

7.1.1. TEXTUAL PRESENTATION. A straightforward and mandatory help is to display both source and trace at the same time, emphasizing their connection. For example, source-oriented tracers move pointers in the source code to show where the execution currently is [154, 156, 29]. Source code, therefore, gives some context to understand execution traces. A next step is to use hypertexts as in the thesis of Calejo [27, chapter 6].

7.1.2. GRAPHICAL PRESENTATION. Quite a number of graphical paradigms exist. These paradigms are at the end of the chain; they do not help any automation, but are instead aimed at users. Enhanced proof trees have been used in the context of sequential systems. OR-parallel systems usually rely on representations of the search tree. Concurrent logic systems favor process-oriented views. Some representations are static, giving a global view, or *signature*, of the execution. Some change over time, leading to either *histories* or *animations*. There is actually no best representation; each representation gives a different view, and is useful in some cases and not so useful in others. Elaborate systems combining different representations, together with new tools based on more powerful analyses, are emerging.

*Execution trees and graphs.* In the context of sequential systems, most graphical systems are based on proof tree representations, which give an immediate picture of the proof structure and are easy to match to the clause structure of the source code, as well as to the implementation. The problem with proof trees, however, is that the information related to backtracking is lost. Ferguson diagrams [184, 50] and Aorta diagrams of the Transparent Prolog machine [71] are enhanced proof trees which retain part of the backtracking information.

In parallel systems, the focus is more on the amount and exploitation of the available parallelism. In OR-parallel systems, the focus has therefore been on the incremental building of the search tree in WAMTRACE [53], Must [177], and Par-Trace [55]. Alhough a proof tree (or procedure invocation tree) can be used to represent an AND-parallel execution, as in VISTA [181], a graph structure has to be used to represent the synchronization induced by the parallel execution of AND branches. VisAndOr [33] uses such a structure in a generic way; it has been designed in order to accommodate different basic execution models (OR-parallelism, restricted AND-parallelism, and determinate-dependent AND-parallelism), as well as to explore representations for combinations of these models.

The static call graph is another means used to graphically give some context to a computation. In [112], Lazzeri first builds a static call graph and, as source-oriented tracers, points to nodes of this graph as the execution proceeds.

*A process-oriented view.* Switching from the standard left to right computation rule of Prolog to coroutining [140] introduces the idea of goals "communicating" data back and forth, and suggests visualizing a logic program as a dynamic network of processes communicating via shared variables, i.e., visualizing the so-called process reading of logic programs [186]. Such a representation has been mainly used in the context of concurrent logic languages [74, 183, 127, 41].

*Histories, animations, and program signatures.* There are many ways to look at the above-mentioned representations. Two basic choices are between a dynamic versus a static view and a local versus a global view. Each view has its advantages.

For instance, the *animation* of the search tree offered by OR-parallel systems is a sophisticated browsing mechanism which turns out to be very good for observing the instantaneous state of a system, but is not able to capture patterns of behavior over time.

Such global patterns can be better analyzed from static *program signatures* such as the ones generated by VisAndOr and VISTA. An interesting feature of VISTA is that procedure-invocation trees of concurrent logic programs are displayed by using radial coordinates and

condensation. This indeed makes it possible to better use the space on the screen, and gives global pictures of executions which are still reasonably "readable" in spite of the number of nodes displayed.

In [41], Conlon and Gregory present a set of local views on query variables, shared variables between processes (channels), and processes, which facilitates the debugging of Parlog programs. In particular, the instantiation of shared variables can be displayed either in *film* format or in *snapshot* format. The film format is an animation of the successive variable instantiations. As animation in general, it does not make it easy to follow the history of the variable over time. When this is necessary, the snapshot format, listing a history of the variable states, can be used. The same kind of idea is also applied to tracing individual processes.

A better representation for parallel and concurrent executions is a process (or agent)-oriented view. The sequential parts are abstracted away and the communication parts are emphasized.

*Integration and abstraction.* More powerful environments can be built by combining the above-mentioned visualizations and better integrating them in the overall environment. For instance, the programming environment of the OR-parallel system MUSE integrates, together with a built-in statistics and benchmarking package, animation and global view by combining Must and VisAndOr [101]. This direction is also followed by its competitor system, Aurora [126]. For the time being, visualization is performed, in both systems, off-line. A further step would be to use, as suggested above, execution replay techniques to synchronize visualization and an execution replay.

A key to a better use of visualization is also the provision of higher-level analyses. This is illustrated by the work on ParSee [108] which, in the context of performance debugging, is able to characterize a predicate by a single colored line combining three abstract measures related to granularity and scheduling.

*Graphical languages.* Following strict visual requirements, a graphical representation of Prolog predicates is used as a basis for a graphical language [109]. It can be animated [160].

Also based on predicate representation, a graphical programming language is underway for concurrent constraint logic programming [99]. It is used for writing, animating, and debugging programs.

## 7.2. Application-Dependent Visualization

As with explanations, the main issue of visualization is actually to support the end-users of the applications. We believe that the problem is not so much to provide nice representations at the level of the programming language, but to provide customization mechanisms and basic graphical packages to help programmers provide appropriate graphic representations of their applications. A survey of the topic can be found in [137].

For example, the KEATS system provides graphical explanations for knowledge-based systems [54]. In [37], mechanisms are introduced to enable customization of the graphics. In [15, 107], parameterization is a general design concept.

Graphical packages that have a proper interface to the environments are essential. Two such packages for Prolog are PCE [194, 195] and the graphic facilities of YAP [113, 114].

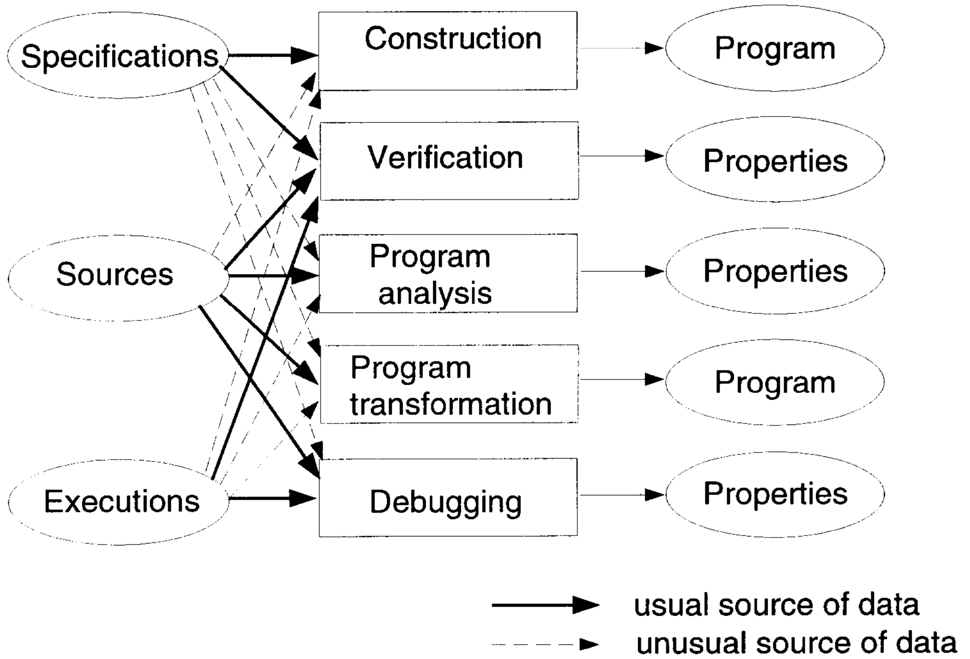


**FIGURE 4.** Current tools waste resources.

## 8. GENERAL PROGRAMMING ENVIRONMENT ISSUES

In this section, we discuss two general problems, namely, integration mechanisms and evaluation criteria.

### *8.1. Integration Mechanisms*

A major issue in programming environments in general (not only for logic programming) is to design integration mechanisms. How can we integrate, in a user-friendly and sound way, so many tools based on so many techniques? Part of the problem is that, at the moment, not all the sources of data are used by the tools. As illustrated by Figure 4, (partial) specifications, source code, and (traces of) executions are not used at their full potential. The three sources of data about a program should be used in synergy if a strategy requires it. The problem is also that tools are artificially partitioned, and that they do not benefit from each other. Users want more than a list of tools, as illustrated by Figure 5.

Regarding integration, one very important requirement is flexibility. As programming environment tools are aiming at accommodating human weaknesses, they have to be flexible. No two people are the same and need exactly the same support at the same time. Some elements of solution are proposed in [60]. The analysis component provides a programming language, namely, Prolog, so that static and dynamic analysis can be jointly programmed. These programs can be neatly integrated into the debugging environment thanks to an extension handler which ensures some consistency in the visibility of the extensions. In particular, on-line help and insertion in the graphical interfaces are provided (almost) for

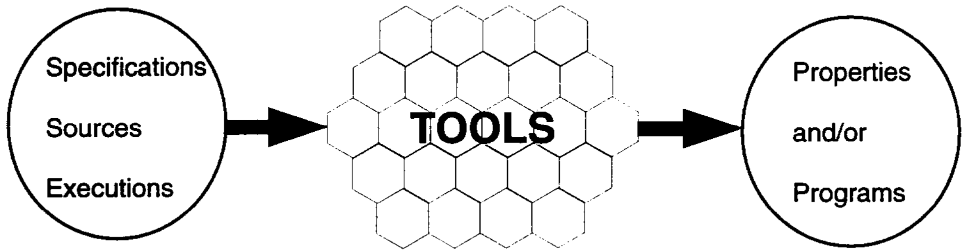


FIGURE 5. Ideal view of programming environments.

free. However, much work remains to be done. The integration mechanisms of the FIELD environment for Unix should give some hints [158].

### 8.2. Evaluation and Validation

It is difficult to draw evaluation criteria of programming environments. Since the tools are user oriented, it is not enough to measure their performance. What should be measured is their usability. The ideal way is to set up inquiries as done by Bergantz and Hassel [11] and Mulholland [136]. Such measurements are very costly. They are nevertheless very instructive. Cooperation with cognitive scientists should be reinforced to try to establish a list of criteria for the evaluation of (logic) programming environments.

Cognitive validation is a long-term objective. In the meantime, a common basis should be established to test and validate the ideas at least on the same ground. In the same way that the implementation community benefited from the setting up and circulating of benchmark suites, a library of important buggy and correct programs should be set up and shared among the logic programming community.

## 9. CONCLUSION

In this survey, we have given a focused and maybe unusual point of view of logic programming environments. A large number of references related to logic programming environments are gathered, which show that the field, although maybe scattered, is active. We have given a framework which separates environment tools into extraction, analysis, and visualization. It emphasizes that *program analysis*, both static and dynamic, is the central issue in programming environments. This point of view helps to analyze existing tools and should give some guidelines for forthcoming ones.

We have listed the achievements in logic programming, pointed out some techniques developed for other languages which should be adapted to logic programming environments, and drawn some trends for further research. Among the main achievements are algorithmic debugging, tracing for sequential Prolog, and abstract interpretation. The main techniques still missing are slicing, test case generation, and program mutation. The perspectives we see are integration, evaluation, and above all, automated static and dynamic analysis.

Programming environments are essential, however good a programming language. As program analysis is the key issue of programming environments, languages with a clean

semantics which enables analysis have an indisputable superiority in the long term. Logic programming languages have this advantage: even Prolog with its well-known weaknesses is better than imperative languages in this area. Environments are therefore a major argument in favor of logic programming. Research on logic programming environments should therefore be mobilized as part of the marketing effort required for logic programming [185].

---

The authors would like to thank Joachim Schimpf for his input on the "Compiled code instrumentation" section. Steven Prestwich helped with the partial evaluation issues. Pascal Brisset, Tony Kusalik, Olivier Ridoux, Mark Wallace, and the anonymous referees gave fruitful comments on this and earlier drafts of the paper. Luke Hornof and Steven Prestwich helped with the English.

The idea of using "double quotes" to implement a simple ground representation was suggested by Lee Naish in a discussion in the newsgroup `comp.lang.prolog`.

The Prolog library of the AIAI at Edinburgh can be retrieved by anonymous ftp (`aiai.edinburgh.ac.uk`).

This work was partly done while the authors were at the European Computer-Industry Research Centre, Munich, Germany. During this period, Mireille Ducassé was partly supported by the ESPRIT Project 5291 CHIC.

---

## REFERENCES

1. Abramson, H., and Rogers, M. H. (eds.), *Meta-Programming in Logic Programming, META'88*, MIT Press, 1989.
2. Aikawa, S., Kamiko, M., Kubo, H., Matsuzawa, F., and Chikayama, T., ParaGraph: A Graphical Tuning Tool for Multiprocessor Systems, in: *FGCS'92* [78], pp. 286–293.
3. ESPRIT P973 ALPES Final report, New University of Lisbon, Portugal, 1989.
4. Arora, T., Ramakrishnan, R., Roth, W. G., Seshadri, P., and Srivastava, D., Explaining Program Execution in Deductive Systems, in: Ceri, Tanaka, and Tsur (eds.), *Proceedings of the Deductive and Object-Oriented Databases Conference*, no. 760 in *Lecture Notes in Computer Science*, Springer-Verlag, Dec. 1993.
5. Av-Ron, E., Top-Down Diagnosis of Prolog Programs, Master's thesis, Weizmann Institute of Science, Israel, 1984.
6. Ball, T., and Horwitz, S., Slicing Programs with Arbitrary Control Flow, in: Fritzson [80], *AADEBUG'93*.
7. Barker-Plummer, D., Cliché Programming in Prolog, in: Bruynooghe [20], pp. 247–271, *META'90*.
8. Barklund, J., What is a Meta-Variable in Prolog? in: Abramson and Rogers [1], pp. 383–398, *META'88*.
9. Baron, U., Chassin de Kergommeaux, J., Hailperin, M., Ratcliffe, M., Robert, P., Syre, J.-C., and Westphal, H., The Parallel ECRC Prolog System PEPSys: An Overview and Evaluation Results, in: *FGCS'88* [77], pp. 841–850.
10. Belli, F., and Jack, O., A Product Assurance Environment for Prolog, in: Ducassé *et al.* [63], *LPE'93*.
11. Bergantz, D., and Hassell, J., Information Relationships in Prolog Programs: How do Programmers Comprehend Functionality? *International Journal of Man-Machine Studies* 35:313–328 (1991), Academic Press.
12. Binks, D., Declarative Debugging of Abstract Data Types in Gödel, in: Fritzson [80], *AADEBUG'93*.
13. Bougé, L., Choquet, N., Fribourg, L., and Gaudel, M.-C., Test Sets Generating from Algebraic Specifications Using Logic Programming, *Journal of System and Software* 6(4):343–360 (1986).

14. Bowles, A. W., and Wilk, P. F., Tracing Requirements for Multi-Layered Meta-Programming, in: Abramson and Rogers [1], pp. 383–398, *META'88.*
15. Brayshaw, M., An Architecture for Visualizing the Execution of Parallel Logic Programs, in: *Proceedings of the International Joint Conference in Artificial Intelligence,* Morgan Kaufmann, 1991, pp. 870–876.
16. Brayshaw, M., and Eisenstadt, M., Adding Data and Procedure Abstraction to the Transparent Prolog Machine (TPM), in: Kowalski and Bowen [106], pp. 532–547, *JICSLP'88.*
17. Breuer, P., Reverse Engineering in Prolog, in: Fuchs *et al.* (ed.), *Logic Programming in Action,* vol. 636 of *Lecture Notes in Artificial Intelligence,* Springer-Verlag, 1992, pp. 290–302.
18. Bronsard, F., Lakshman, T. K., and Reddy, U. S., A Framework of Directionality for Proving Termination of Logic Programs, in: Apt (ed.), *Proceedings of the Joint International Conference and Symposium on Logic Programming,* Washington, DC, Nov. 1992, pp. 321–335, MIT Press, *JICSLP'92.*
19. Bruffaerts, A., and Henin, E., Proof Trees for Negation as Failure: Yet Another Prolog Meta-Interpreter, in: Kowalski and Bowen [106], pp. 343–358, *JICSLP'88.*
20. Bruynooghe, M. (ed.), *Proceedings of the Second Workshop on Meta-Programming in Logic,* Department of Computer Science, K.U. Leuven, Belgium, 1990, META'90.
21. Bruynooghe, M., A Practical Framework for the Abstract Interpretation of Logic Programs, *Journal of Logic Programming* 10(2):91–124 (1991).
22. Bruynooghe, M., Special Issue on Abstract Interpretation, *Journal of Logic Programming,* 13 (July 1992).
23. Bruynooghe, M., and Janssens, G., An Instance of Abstract Interpretation Integrating Type and Mode Inference, in: Kowalski and Bowen [106], pp. 669–683, *JICSLP'88.*
24. Bruynooghe, M., and Wirsing, M. (eds.), *International Symposium on Programming Language Implementation and Logic Programming,* vol. 631 of *Lecture Notes in Computer Science, PLILP'92,* Springer-Verlag, Aug. 1992.
25. Bundy, A., Pain, H., Brna, P., and Lynch, L., A Proposed Prolog Story, D. A. I. Research Paper 283, Department of Artificial Intelligence, University of Edinburgh, 1986.
26. Byrd, L., Understanding the Control Flow of Prolog Programs, in: Tärnlund (ed.), *Logic Programming Workshop,* Debrecen, Hungary, 1980.
27. Calejo, M. C., A Framework for Declarative Prolog Debugging, Ph.D. thesis, New University of Lisbon, Portugal, Mar. 1992.
28. Calejo, M. C., and Pereira, L. M., Declarative Source Debugging, in: *Proceedings of the 5th Portuguese AI Conference,* vol. 541 of *Lecture Notes in Computer Science,* Springer-Verlag, 1991.
29. Callebaut, A., and Demoen, B., Program Sources as Model for Debugging in Prolog, in: Ducassé *et al.* [64], *LPE'90.*
30. Campagnoni, F. R., and Ehrlich, K., Information Retrieval Using a Hypertext-Based Help System, *SIGIR Forum,* June 1989, pp. 212–220, *Proceedings of the 12th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval.*
31. Canfora, G., Cimitile, A., and de Carlini, U., A Logic-Based Approach to Reverse Engineering Tools Production, *IEEE Transactions on Software Engineering* 18(12):1053–1064 (Dec. 1992).
32. Carlsson, M., Lusk, E., and Szeredi, P., Smoothing Rough Edges in Aurora, in: *Proceedings of the First COMPULOG-NOE Area Meeting on Parallelism and Implementation Technology,* Technical University of Madrid, Spain, May 1993.
33. Carro, M., Gomez, L., and Hermenegildo, M., Some Paradigms for Visualizing Parallel Execution of Logic Programs, in: D. S. Warren (ed.), *Proceedings of the International Conference on Logic Programming,* Budapest, Hungary, 1993, MIT Press.
34. Casson, A., Event Abstraction Debuggers for Layered Systems in Prolog, in: *Proceedings of the UK Logic Programming Conference,* Association for Logic Programming, UK Branch, Mar. 1990.

35. Cheng, J., Slicing Concurrent Programs, in: Fritzson [80], *AADEBUG'93*.
36. Cherniak, C., Undebuggability and Cognitive Science, *Communications of the ACM* 31(4):402–412 (Apr. 1988).
37. Cochard, J.-L., A Graphical Representation Environment for Complex Prolog Structures, in: Ducassé *et al.* [64], *LPE'90*.
38. Codish, M., Dams, D., and Shapiro, E., Automatic Detection of Reply Variables in Concurrent Logic Programs, in: Bruynooghe [20], pp. 325–338, *META'90*.
39. Codognet, C., Codognet, P., and Corsini, M., Abstract Interpretation of Concurrent Logic Programs, in: Debray and Hermenegildo [45], pp. 215–232, *NACLP'90*.
40. Cohen, J., and Carpenter, N., A Language for Inquiring about the Run-Time Behavior of Programs, *Software—Practice and Experience* 7:445–460 (1977).
41. Conlon, T., and Gregory, S., Debugging Tools for Concurrent Logic Programming, *The Computer Journal* 35(2):157–169 (1992), Wiley.
42. Coombs, M. J., and Alty, J., Expert Systems: An Alternative Paradigm, in: Coombs (ed.), *Developments in Expert Systems*, Academic Press, London, 1984, pp. 135–157.
43. Coombs, M. J., Hartley, R. T., and Stell, J. G., Debugging User Conceptions of Interpretation Processes, in: *Proceedings of the AAAI'86*, Morgan Kaufmann, 1986, pp. 303–307.
44. Chassin de Kergommeaux, J., Peterson, D., Rapp, W., and Westphal, H., The Implementation of PEPSys on a MX-500 Multiprocessor, Technical Report CA-38, ECRC, 1988.
45. Debray, S. K., and Hermenegildo, M. (eds.), *Proceedings of the North American Conference on Logic Programming, NACLP'90*, Austin, TX, MIT Press, 1990.
46. Debray, S. K., Static Inference of Modes and Data Dependencies in Logic Programs, *ACM Transactions on Programming Languages and Systems* 11(3):418–450 (July 1989).
47. DeMillo, R. A., McCracken, W. M., Martin, R. J., and Passafiume, J. F., *Software Testing and Evaluation*, Benjamin/Cumming, Menlo Park, 1987.
48. Dershowitz, N., and Lee, Y., Deductive Debugging, in: *SLP'87* [171], pp. 298–306.
49. Deville, Y., and Lau, K.-K., Logic Program Synthesis, *Journal of Logic Programming* 19, 20:321–350 (1994). Also, Research Report RR 93-19, Université Catholique de Louvain, Unité d'Informatique.
50. Dewar, A. D., and Cleary, J. G., Graphical Display of Complex Information within a Prolog Debugger, *International Journal of Man-Machine Studies* 25(5):503–521 (Nov. 1986), Academic Press.
51. Diehl, C., Jard, C., and Rampon, J.-X., Reachability Analysis on Distributed Executions, in: Gaudel and Jouannaud (eds.), *Proceedings of the International Conference on Theory and Practice of Software Development (TAPSOFT'93)*, vol. 668 of *Lecture Notes in Computer Science*, Springer-Verlag, 1993, pp. 629–643.
52. Dincbas, M., Van Hentenryck, P., Simonis, H., Aggoun, A., Graf, T., and Berthier, F., The Constraint Logic Programming Language CHIP, in: *FGCS'88* [77].
53. Disz, T., and Lusk, E., A Graphical Tool for Observing the Behavior of Parallel Logic Programs, in: *SLP'87* [171], pp. 46–53.
54. Domingue, J., and Eisenstadt, M., A New Metaphor for the Graphical Explanation of Forward-Chaining Rule Execution, in: *Proceedings of the International Joint Conference on Artificial Intelligence*, Morgan Kaufmann, 1989, pp. 129–134.
55. Dorochevsky, M., and Xu, J., ElipSys Parallel Execution Tracer, Internal Report 91-7i, ECRC, Feb. 1992.
56. Drabent, W., Nadjm-Tehrani, S., and Maluszynski, J., The Use of Assertions in Algorithmic Debugging, in: *FGCS'88* [77], pp. 573–581.
57. Drabent, W., Nadjm-Tehrani, S., and Maluszynski, J., Algorithmic Debugging with Assertions, in: Abramson and Rogers [1], pp. 383–398, *META'88*.
58. Ducassé, M., Abstract Views of Prolog Executions in Opium, in: Saraswat and Ueda [162], pp. 18–32, *ILPS'91*.
59. Ducassé, M., Analysis of Failing Prolog Executions, in: *Proceedings of the ICLP'91*

*Workshop on Logic Programming Environments*, Paris, June 1991, Technical Report LIFO N 91-61, University of Orléans, France.
60. Ducassé, M., An Extendable Trace Analyser to Support Automated Debugging, Ph.D. thesis, University of Rennes I, France, June 1992, European Doctorate.
61. Ducassé, M., A General Trace Query Mechanism Based on Prolog, in: Bruynooghe and Wirsing [24], pp. 400–414, *PLILP'92*.
62. Ducassé, M., A Pragmatic Survey of Automated Debugging, in: Fritzson [80], *AADE-BUG'93*.
63. Ducassé, M., Le Charlier, B., Lin, Y.-J., and Yalcinalp, U. (eds.), *Proceedings of ILPS'93 Workshop on Logic Programming Environments, LPE'93*, Vancouver, Oct. 1993, Publication IRISA, Campus de Beaulieu, 35042 Rennes, France,
64. Ducassé, M., Emde, A.-M., Kusalik, T., and Levy, J. (eds.), *Proceedings of ICLP'90 Workshop on Logic Programming Environments, LPE'90*, Eilat, June 1990, Technical Report, ECRC IR-LP-31-25, European Computer-Industry Research Centre.
65. Ducassé, M., and Ferrand, G. (eds.), *Proceedings of ICLP'91 Workshop on Logic Programming Environments, LPE'91*, Paris, June 1991, Technical Report, University of Orléans, France, LIFO N 91-61.
66. Ducassé, M., Lin, Y.-J., and Yalcinalp, L. Ü. (eds.), *Proceedings of IJCSLP'92 Workshop on Logic Programming Environments, LPE'92*, Washington, Nov. 1992, Technical Report TR 92-143, Case Western Reserve University, Cleveland.
67. Ducassé, M., and Noyé, J., Tracing Compiled Prolog Code, Publication IRISA, In preparation, 1994.
68. Duesterwald, E., Gupta, R., and Soffa, M. L., Distributed Slicing and Partial Reexecution for Distributed Programs, in: *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, vol. 589 of *Lecture Notes in Computer Science*, Springer-Verlag, 1992.
69. Eisenstadt, M., A Powerful Prolog Trace Package, in: *Proceedings of the 6th European Conference on Artificial Intelligence*, North-Holland, 1985, Sept. 1984.
70. Eisenstadt, M., Retrospective Zooming, A Knowledge Based Tracing and Debugging Methodology for Logic Programming, in: *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, Morgan Kaufmann, 1985.
71. Eisenstadt, M., and Brayshaw, M., The Transparent Prolog Machine (TPM): An Execution Model and Graphical Debugger for Logic Programming, *Journal of Logic Programming* 5(4):277–342 (1988).
72. Emde, A.-M., and Ducassé, M., Automated Debugging of Non-Terminating Prolog Programs, in: *Proceedings of the ICLP'90 Workshop on Logic Programming Environments*, June 1990, Technical Report, ECRC IR-LP-31-25.
73. Eriksson, A., and Johansson, A.-L., Neat Explanation of Proof Trees, in: *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, Morgan Kaufmann, 1985, pp. 379–381.
74. Feldman, Y., and Shapiro, E., Temporal Debugging and Its Visual Animation, in: Saraswat and Ueda [162], pp. 3–17, *ILPS'91*.
75. Fernández, M. J., Carro, M., and Hermenegildo, M., IDeal Resource Allocation (IDRA): A Technique for Computing Accurate Ideal Speedups in Parallel Logic Languages, Technical Report FIM26.3/AI/92, Computer Science Faculty, Technical University of Madrid, Sept. 1992.
76. Ferrand, G., Error Diagnosis in Logic Programming, An Adaptation of E.Y. Shapiro's Method, *Journal of Logic Programming* 4:177–198 (1987).
77. *Proceedings of the International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, Dec. 1988, ICOT.
78. *Proceedings of the International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, June 1992, ICOT.

79. Fidge, C., Logical Time in Distributed Computing Systems, *IEEE Computer* 28–33 (Aug. 1991).
80. Fritzson, P. (ed.), *Proceedings of the First Workshop on Automated and Algorithmic Debugging*, vol. 749 of *Lecture Notes in Computer Sciences*, Linkoeping, Sweden, May 1993, Springer-Verlag, *AADEBUG'93*.
81. Fromherz, M. P. J., Declarative Debugging of Concurrent Constraint Programs, in: Fritzson [80], *AADEBUG'93*.
82. Fuchs, N. E., Specifications Are (Preferably) Executable, *Software Engineering Journal* 323–334 (Sept. 1992), IEE Publications. Also, Technical Report 91.10, Department of Computer Science, University of Zurich.
83. Gaifman, H., Maher, M. J., and Shapiro, E., Replay, Recovery, Replication and Snapshots of Nondeterministic Concurrent Programs, in: *Proceedings of the International Conference in Principles of Distributed Computing*, Montreal, Canada, 1991, ACM Press.
84. Gallagher, J., Transforming Logic Programs by Specialising Interpreters, in: *7th European Conference on Artificial Intelligence*, North-Holland, 1986, pp. 109–122.
85. Gorlick, M. M., and Kesselman, C. F., Timing Prolog Programs without Clocks, in: *SLP'87* [171], pp. 426–432.
86. Gorlick, M. M., Kesselman, C. F., Marotta, D. A., and Parker, D. S., Mockingbird: A Logical Methodology for Testing, *The Journal of Logic Programming* 8(1 and 2):95–120 (Jan./Mar. 1990).
87. Graham, S. L., Kessler, P. B., and McKusick, M. K., gprof: A Call Graph Execution Profiler, in: *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, June 1982, pp. 120–126, *SIGPLAN Notices* 17(6).
88. Hanus, M., and Josephs, B., A Debugging Model for Functional Logic Programs, in: M. Bruynooghe and J. Penjam (eds.), *Proceedings of the Programming Language Implementation and Logic Programming Symposium*, vol. 714 of *Lecture Notes in Computer Science*, Springer-Verlag, 1993.
89. Hermenegildo, M. V., Abstract Interpretation and Its Applications, Tutorial presented at JICSLP'92, Nov. 1992, Technical University of Madrid, Spain.
90. Hill, P. M., and Lloyd, J. W., The Gödel Programming Language, Technical Report CSTR-92-27, Department of Computer Science, University of Bristol, Oct. 1992.
91. Hill, P. M., and Lloyd, J. W., Analysis of Meta-Programs, in: Abramson and Rogers [1], pp. 23–51, *META'88*.
92. Hirunkitti, V., and Hogger, C. J., A Generalised Query Minimisation for Program Debugging, in: Fritzson [80], *AADEBUG'93*.
93. Hoffman, D., and Strooper, P., Automated Module Testing in Prolog, *IEEE Transactions on Software Engineering* 17(9):934–943 (Sept. 1991).
94. Höök, K., Taylor, J., and du Boulay, B., Redo "TRY ONCE and PASS": The Influence of Complexity and Graphical Notation on Novices' Understanding of Prolog, *Instructional Science* 19:337–360 (1990), Kluwer Academic Publishers.
95. Höök, K., Wærn, A., and Pain, H., Possible Extensions to the Byrd Box Tracer Aimed at Experts, Poster presented at the Workshop of the Psychology of Programming Interest Group, UK, Jan. 1992.
96. Huntbach, M. M., Algorithmic Parlog Debugging, in: *SLP'87* [171], pp. 288–297.
97. Inamura, Y., and Onishi, S., A Detection Algorithm of Perpetual Suspension in KL1, in: Warren and Szeredi [191], pp. 18–30, *ICLP'90*.
98. Ishida, S., and Chikayama, T., Programming Environment of PIMOS, in: Ducassé *et al.* [66], pp. 48–54, *LPE'92*.
99. Kahn, K. M., Concurrent Constraint Programs to Parse and Animate Pictures of Concurrent Constraint Programs, in: *FGCS'92* [78].

100. Kamkar, M., Shahmehri, N., and Fritzson, P., Bug Localization by Algorithmic Debugging and Program Slicing, in: Deransart and Maluszynski (eds.), *Proceedings of the Programming Language Implementation and Logic Programming Symposium*, vol. 456 of *Lecture Notes in Computer Science*, Linkoeping, Sweden, Aug. 1990, Springer-Verlag.
101. Karlsson, R., A High-Performance OR-Parallel System, Ph.D. thesis, The Royal Institute of Technology, Stockholm, Sweden, Mar. 1992.
102. Klint, P., Reps, T., and Snelting, G., Programming Environments, Report on an International Workshop at Dagstuhl Castle, *ACM SIGPLAN Notices* 27(11):90–96 (Nov. 1992).
103. Komorowski, J., Partial Evaluation as a Means for Inferencing Data Structures in an Applicative Language: A Theory and Implementation in the Case of Prolog, in: *Proceedings of the Symposium on Principles of Programming Languages*, ACM, 1982, pp. 255–267.
104. Komorowski, J. (ed.), *Proceeedings of the Workshop on Prolog Programming Environments*, Sweden, 1982, University of Linkoeping.
105. Korel, B., Identifying Faulty Modifications in Software Maintenance, in: Fritzson [80], *AADEBUG'93*.
106. Kowalski, R. A., and Bowen, K. A. (eds.), *Proceedings of the Joint International Conference and Symposium on Logic Programming, JICSLP'88*, Seattle, WA, Aug. 1988, MIT Press.
107. Kusalik, A. J., and Oster, G. M., Towards a Generalized Graphical Interface for Logic Programming Development, Technical Report 93-2, University of Saskatchewan, Canada, 1993.
108. Kusalik, A. J., and Prestwich, S. D., Programmer-Oriented Visualisation of Parallel Logic Program Execution, in: Ducassé *et al.* [63], *LPE'93*.
109. Ladret, D., and Rueher, M., Vlp: A Visual Logic Programming Language, *Journal of Visual Languages and Computing* 2:163–188 (1991), Academic Press.
110. Lakshman, T. K., and Reddy, U. S., Typed Prolog: A Semantic Reconstruction of the Mycroft-O'Keefe Type System, in: Saraswat and Ueda [162], pp. 202–220, *ILPS'91*.
111. Lamport, L., Time, Clocks, and the Ordering of Events in a Distributed System, *Communications of the ACM* 21(7):558–565 (July 1978).
112. Lazzeri, S. G., Vizzprol: A Tool for Visualizing Prolog Programs, in: Ducassé and Ferrand [65], *LPE'91*.
113. Leal, J. P., The Ytoolkit: A Prolog Approach to a User Interface, in: Ducassé *et al.* [64], *LPE'90*.
114. Leal, J. P., Damas, L., and Moreira, N., An History Based Interface, in: Ducassé and Ferrand [65], *LPE'91*.
115. Leblanc, T., and Mellor-Crummey, J., Debugging Parallel Programs with Instant Replay, *IEEE Transactions on Computers* C-36(4):471–481 (Apr. 1987).
116. Lee, Y.-J., and Dershowitz, N., Debugging Logic Programs Using Specifications, in: Fritzson [80], *AADEBUG'93*.
117. Leu, E., and Schiper, A., Execution Replay: A Mechanism for Integrating a Visualization Tool With, in: Bougé, Cosnard, Robert, and Trystram (eds.), *Proceedings of the 2nd Joint International Conference on Vector and Parallel Processing*, vol. 634 of *Lecture Notes in Computer Science*, Springer-Verlag, Sept. 1992, pp. 55–66.
118. Leu, E., Schiper, A., and Zramdini, A., Efficient Execution Replay Technique for Distributed Memory Architectures, in: Bode (ed.), *Proceedings of the 2nd European Conference on Distributed Memory Computing*, vol. 487 of *Lecture Notes in Computer Science*, Springer-Verlag, 1991, pp. 315–324.
119. Levi, G., and Martelli, M. (eds.), *Proceedings of the International Conference on Logic Programming, ICLP'89*, Lisbon, Portugal, June 1989, MIT Press.
120. Lichtenstein, Y., and Shapiro, E., Abstract Algorithmic Debugging, in: Kowalski and Bowen [106], *JICSLP'88*.
121. Lim, P., and Stuckey, P. J., Meta Programming as Constraint Programming, in: Debray and Hermenegildo [45], pp. 416–430, *NACLP'90*.

122. Lingenfelder, C., Structuring Computer Generated Proofs, in: *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, Morgan Kaufmann, 1989, pp. 378–383.
123. Lloyd, J. W., Declarative Error Diagnosis, *New Generation Computing* 5(2):133–154 (1987), Springer-Verlag.
124. Lloyd, J. W., Directions for Meta-Programming, in: *FGCS'88* [77], pp. 609–617.
125. Looi, C.-K., Analysing Novices' Programs in a Prolog Intelligent Teaching System, in: *Proceedings of the European Conference on Artificial Intelligence*, Munich, Aug. 1988, pp. 314–319, Pitman.
126. Lusk, E., Mudambi, S., Overbeek, R., and Szeredi, P., Applications of the Aurora Parallel Prolog System to Computational Molecular Biology, in: Miller (ed.), *Proceedings of the International Logic Programming Symposium, ILPS'93*, Vancouver, Canada, Oct. 1993, pp. 353–369, MIT Press.
127. Maeda, M., Implementing a Process Oriented Debugger with Reflection and Program Transformation, in: *FGCS'92* [78], pp. 961–968.
128. Malony, A., Hammerslag, D., and Jablonowski, D., Traceview: A Trace Visualization Tool, *IEEE Software* 19–28 (Sept. 1991).
129. Marre, B., Toward Automatic Test Data Set Selection Using Algebraic Specifications and Logic Programming, in: Furukawa (ed.), *Proceedings of the International Conference on Logic Programming*, Paris, France, 1991, pp. 202–219, MIT Press.
130. McDowell, C. E., and Helmbold, D. P., Debugging Concurrent Programs, *ACM Computing Surveys* 21(4):593–622 (Dec. 1989).
131. Meier, M., Aggoun, A., Chan, D., Dufresne, P., Enders, R., Henry de Villeneuve, D., Herold, A., Kay, P., Perez, B., van Rossum, E., and Schimpf, J., SEPIA—An Extendible Prolog System, in: *Proceedings of the IFIP '89*, 1989.
132. Mellish, C., Abstract Interpretation of Prolog Programs, in: Abramsky and Hankin (eds.), *Abstract Interpretation of Declarative Languages*, Ellis Horwood Limited, 1987, ch. 8, pp. 181–198.
133. Miller, G. A., The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information, *Psychological Review* 63:81–97 (1956).
134. Monteiro, L., and Porto, A., Contextual Logic Programming, in: Levi and Martelli [119], pp. 284–299, *ICLP'89*.
135. Moroshita, S., and Numao, M., Prolog Computation Model BPM and Its Debugger PROEDIT2, in: *Proceedings of the 5th Logic Programming Conference*, Tokyo, June 1986, pp. 147–158, Springer-Verlag.
136. Mulholland, P., The Effect of Graphical and Textual Visualisation on the Comprehension of Prolog Execution by Novices: An Empirical Analysis, Human Cognition Research Laboratory, Open University, Milton Keynes, UK, 1993.
137. Murray, B. S., and McDaid, E., Visualizing and Representing Knowledge for the End User: A Review, *International Journal of Man-Machine Studies* 38:23–49 (1993), Academic Press.
138. Myers, G. J., The Art of Software Testing, in: *Business Data Processing*, Wiley, 1979, ISBN 0-471-04328-1.
139. Nadathur, G., and Miller, D. A., An Overview of λProlog, in: Kowalski and Bowen [106], pp. 810–827, *JICSLP'88*.
140. Naish, L., *Negation and Control in Prolog*, vol. 238 of *Lecture Notes in Computer Science*, Springer-Verlag, 1986.
141. Naish, L., Declarative Debugging of Lazy Functional Programs, in: Ducassé *et al.* [66], pp. 29–34, *LPE'92*.
142. Naish, L., Types and the Intended Meaning of Logic Programs, in: Pfenning [153], ch. 6, pp. 189–216, ISBN 0-262-16131-1.
143. Naish, L., Dart, P. W., and Zobel, J., The NU-Prolog Debugging Environment, in: Levi and Martelli [119], *ICLP'89*.

144. Nakashima, H., Tomura, S., and Ueda, K., What is a Variable in Prolog? in: *Proceedings of the International Conference on Fifth Generation Computer Systems*, ICOT, 1984, pp. 327–332.
145. Nilsson, H., and Fritzson, P., Algorithmic Debugging of Lazy Functional Languages, in: Bruynooghe and Wirsing [24], pp. 385–399, *PLILP'92*.
146. Noyé, J., An Overview of the Knowledge Crunching Machine, in: Abdelguerfi and Lavington (eds.), *Emerging Trends in Database and Knowledge-base Machines*, IEEE Computer Society Press, 1994.
147. Olsson, R. A., Crawford, R. H., and Ho, W. W., A Dataflow Approach to Event-Based Debugging, *Software-Practice and Experience* 21(2):209–229 (Feb. 1991), Wiley.
148. Osterweil, L., and Clarke, L. A., A Proposed Testing and Analysis Research Initiative, *IEEE Software* 89–96 (Sept. 1992).
149. Paris, C. L., Generation and Explanation: Building an Explanation Facility for the Explainable Expert Systems Framework, in: Paris, Swartout, and Mann (eds.), *Natural Language Generation in Artificial Intelligence and Computational Linguistics*, Kluwer, Boston, 1991, pp. 49–81.
150. Pereira, L. M., Rational Debugging in Logic Programming, in: Shapiro (ed.), *Proceedings of the International Logic Programming Conference*, vol. 225 of *Lecture Notes in Computer Science*, London, UK, July 1986, pp. 203–210, Springer-Verlag.
151. Pereira, L. M., and Calejo, M. C., A Framework for Prolog Debugging, in: Kowalski and Bowen [106], *JICSLP'88*.
152. Pereira, L. M., Damasio, C. V., and Alferes, J. J., Debugging by Diagnosing Assumptions, in: Fritzson [80], *AADEBUG'93*.
153. Pfenning, F. (ed.), *Types in Logic Programming*, MIT Press, 1992, ISBN 0-262-16131-1.
154. Plummer, D., Coda: An Extended Debugger for Prolog, Technical Report AI87-54, University of Texas at Austin, Apr. 1987.
155. Podgurski, A., and Clarke, L. A., The Implications of Program Dependences for Software Testing, Debugging, and Maintenance, in: Kemmerer (ed.), *Proceedings of the Third Symposium on Software Testing, Analysis and Verification (TAV3)*, ACM-SIGSOFT, Dec. 1989, pp. 168–178, *Software Engineering Notes* 14(8).
156. Rajan, T., Apt: A Principled Design of an Animated View of Program Execution for Novice Programmers, in: Bullinger and Shackel (eds.), *Human-Computer Interaction—INTERACT'87*, IFIP, Sept. 1987, pp. 291–296.
157. Raynal, M., About Logical Clocks for Distributed Systems, *Operating Systems Review (ACM SIGOPS)* 26(1):41–48 (Jan. 1992).
158. Reiss, S. P., Connecting Tools Using Message Passing in the FIELD Program Development Environment, *IEEE Software* 57–66 (July 1990).
159. Rousset, M. C., and Safar, B., Negative and Positive Explanations in Expert Systems, *Applied Artificial Intelligence* 1 (1987), Hemisphere Publishing Corporation.
160. Rueher, M., Revisiting Capabilities of Graphic for Logic Programming, in: Ducassé and Ferrand [65], *LPE'91*.
161. Safra, S., and Shapiro, E., Meta Interpreters for Real, in: Kugler (ed.), *Information Processing*, Dublin, Ireland, 1986, pp. 271–278, North-Holland,
162. Saraswat, V., and Ueda, K. (eds.), *Proceedings of the International Logic Programming Symposium, ILPS'91*, San Diego, CA, Oct. 1991, MIT Press.
163. Saurel, C., Contribution aux Systèmes Experts: Développement d'un Cas Concret et Etude du Problème de la Génération d'Explications Négatives, Ph.D. thesis, ENSAE, Toulouse, Dec. 1987.
164. Schleiermacher, A., and Winkler, J. F. H., The Implementation of ProTest, A Prolog Debugger for a Refined Box Model, *Software—Practice and Experience* (1990), Wiley.
165. Sehr, D. C., and Kalé, L. V., Estimating the Inherent Parallelism in Prolog Programs, in: *FGCS'92* [78], pp. 783–790.

166. Shahmehri, N., Generalized Algorithmic Debugging, Ph.D. thesis, Department of Computer Science, Linkoeping University, Sweden, 1991, Dissertation no. 297.
167. Shapiro, E., The Family of Concurrent Logic Programming Languages, *ACM Computing Surveys* 21(3):413–510 (Sept. 1989).
168. Shapiro, E., *Algorithmic Program Debugging*, MIT Press, Cambridge, MA, 1983, ISBN 0-262-19218-7.
169. Shmueli, O., and Tsur, S., Logical Diagnosis of LDL Programs, in: Warren and Szeredi [191], pp. 112–129, *ICLP'90*.
170. Silverman, W., Hirsch, M., Houri, M., and Shapiro, E., The Logix System User Manual, in: Shapiro (ed.), *Concurrent Prolog: Collected Papers*, MIT Press, 1987.
171. *Proceedings of the Symposium on Logic Programming*, San Francisco, CA, Sept. 1987, IEEE Computer Society.
172. Specht, G., Generating Explanation Trees Even for Negations in Deductive Database Systems, in: Ducassé *et al.* [63], *LPE'93*.
173. Sterling, L., and Beer, R. D., Incremental Flavor-Mixing of Meta-Interpreters for Expert System Construction, in: *Proceedings of the 3rd Symposium on Logic Programming*, Salt Lake City, 1986, pp. 20–27.
174. Sterling, L., and Shapiro, E., *The Art of Prolog*, MIT Press, Cambridge, MA, 1986, ISBN 0-262-69105-1.
175. Sterling, L., and Yalcinalp, L. Ü., Explaining Prolog Based Expert Systems Using a Layered Meta-Interpreter, in: *Proceedings of the International Joint Conference on Artificial Intelligence*, Morgan Kaufmann, 1989.
176. Strooper, P., and Hoffman, D., Prolog Testing of C Modules, in: Saraswat and Ueda [162], pp. 596–610, *ILPS'91*.
177. Svensson, C., and Sundberg, J., MUSE Trace. A Graphic Tracer for Or-Parallel Prolog, Technical Report T90003, SICS, Sweden, 1990.
178. Swartout, W. R., Nordin, H., Paris, C., and Smoliar, S. W., Toward a Rapid Prototyping Environment for Expert Systems, in: *Proceedings of the 13th German Workshop on Artificial Intelligence*, Springer-Verlag, 1989.
179. Takahashi, H., and Shibayama, E., Preset—A Debugging Environment for Prolog, in: *Logic Programming Conference*, vol. 221 of *Lecture Notes in Computer Science*, Tokyo, Japan, 1985, pp. 90–99, Springer-Verlag.
180. Takeuchi, A., Algorithmic Debugging of GHC Programs and its Implementation in GHC, in: Shapiro (ed.), *Concurrent Prolog: Collected Papers*, MIT Press, Boston, 1987, pp. 180–196.
181. Tick, E., Visualizing Parallel Logic Programs with VISTA, in: *FGCS'92* [78], pp. 934–942.
182. Tobermann, G., and Beckstein, C., What's in a Trace: The Box Model Revisited, in: Fritzson [80], *AADEBUG'93*.
183. Trehan, R., A Process Based Tracer for KL1 on PIM, in: Ducassé and Ferrand [65], *LPE'91*.
184. van Emden, M. H., An Interpreting Algorithm for Prolog Programs, in: *Proceedings of the First International Conference on Logic Programming*, Marseille, France, Sept. 1982.
185. van Emden, M. H., Marketing for Logic Programming, *Newsletter of the Association for Logic Programming* 6(1):3–4 (Feb. 1993).
186. van Emden, M. H., and de Lucena, G. J., Logic Programming, in: Clark and Tärnlund (eds.), *Logic Programming*, chapter Predicate Logic as a Language for Parallel Programming, Academic Press, 1982, pp. 189–198.
187. van Harmelen, F., Definable Naming Relations in Meta-Level Systems, in: Pettorossi (ed.), *Proceedings of the Meta-Programming in Logic, Third International Workshop, META-92*, Uppsala, Sweden, June 1992, vol. 649 of *Lecture Notes in Computer Science*, Springer-Verlag, 1992, pp. 89–104.

188. van Rossum, E., Implémentation d'un Debugger Prolog, Master's thesis, Facultés Universitaires Notre Dame de la Paix, Namur, Belgique, 1989.
189. Van Someren, M. W., Beginner's Problems in Learning Prolog, Memo 54, University of Amsterdam, 1985.
190. Walker, A., Prolog/Ex1, An Inference Engine Which Explains Both Yes and No Answers, in: *Proceedings of the International Joint Conference on Artificial Intelligence*, Morgan Kaufmann, 1983, pp. 526–528.
191. Warren, D. H. D., and Szeredi, P. (eds.), *Proceedings of the International Conference on Logic Programming, ICLP'90*, Jerusalem, Israel, June 1990, MIT Press.
192. Weiser, M., Program Slicing, *IEEE Transactions on Software Engineering* SE-10(4):352–357 (July 1984).
193. Wieland, C., Two Explanation Facilities for the Deductive Database Management System DeDex, in: Kangassalo (ed.), *Proceedings of the 9th Conference on Entity-Relationship Approach*, 1990, ETH Zurich, pp. 189–203.
194. Wielemaker, J., and Anjewierden, A., *PCE-4 Functional Overview*, SWI, University of Amsterdam, Roetersstraat 15, 1018 WB Amsterdam, The Netherlands, 1992.
195. Wielemaker, J., and Anjewierden, A., *Programming in PCE/Prolog*, SWI, University of Amsterdam, Roetersstraat 15, 1018 WB Amsterdam, The Netherlands, 1992.
196. Yalcinalp, L. Ü., Meta-Programming for Knowledge Based Systems in Prolog, Ph.D. thesis, Case Western Reserve University, Cleveland, OH, Aug. 1991, Technical Report TR 91-141.
197. Yalcinalp, L. Ü., and Sterling, L., An Integrated Interpreter for Explaining Prolog's Successes and Failures, in: Abramson and Rogers [1], *META'88*.