Miguel Cruz Costa Calejo

A Framework for Declarative Prolog Debugging

PhD thesis in Computer Science, Artificial Intelligence branch

Computer Science Department,
Universidade Nova de Lisboa

March 1992

Acknowledgements

First of all, to my family, due to obvious social-biological constraints and specially to *many* other non-obvious reasons. In particular, to my parents and my wife, whose lifes were substantially affected.

Luís Moniz Pereira guided me in the ways of research, passing me one of his pet topics. His continuous tenacity, patience and permanent availability, even amidst lots of bureaucratic turbulence, are perhaps the main reasons for the successful completion of this work. António Porto provided an invaluable and reliable source for rational discussions of any kind along the years; together with Luís he also convinced me to leave my home town environment, and come to the big city where interesting things were happening.

Other colleagues provided reasons for productive discussions, namely Luís Trindade, Luís Caires, Artur Miguel Dias, José Alegria, Salvador Abreu and Paulo Rosado. Joaquim Brigas, Steiger Garção and others helped me settling up in Lisbon.

On the international side, Fabrizio Gagliardi, Michael Poe, and Norbert Fuchs offered me the means to spend time in very interesting places and in good company. Francesco Russo, Mireille Ducassé and Yossi Lichtenstein enriched my mental processes through our discussions.

My former boss, Paulo Vicente, provided me with the opportunity to see the “real world” outside academia, but neverthless forced me to finish this thesis during one of my periods of doubt, even against his company's technical health. My current boss, Luiz Lopes da Silva, offered me paid vacations to write the final version.

My work was supported by Instituto Nacional de Investigação Científica, Centro de Informática da Universidade Nova de Lisboa (Linha de Acção IV), Junta Nacional de Investigação Científica e Tecnológica, Centro de Inteligência Artificial do UNINOVA, the Portuguese CERN Commission, the Advanced Logic Programming Environments Project (ESPRIT project 973), the COMPULOG ESPRIT Basic Research Action, Apple Computer, Inc., Interlog, SA, IBM Portugal, SA, Softlog, SA, and Servisoft, Lda.

And of course, this section violates the closed world assumption: there are many others not mentioned previously to whom I own. Thanks to them as well.

Summary in Portuguese

Neste relatório de tese descreve-se uma nova metodologia para depuração ("debugging") de programas Prolog, assente em informação declarativa prestada pelo utilizador.

Este trabalho constitui um melhoramento às abordagens anteriores no campo de depuração declarativa, a vários níveis: diminuição substancial do número necessário de perguntas ao utilizador; primeiro tratamento específico para impurezas extra-lógicas, como o operador de corte e os efeitos colaterais internos; tratamento uniforme para todos os tipos de erro, excepto a não-terminação; e um protótipo experimental incorporando estes melhoramentos.

Nota: no apêndice A inclui-se um resumo em Português bastante mais alargado, de acordo com o regulamento da FCT/UNL.

Abstract

This thesis describes a new approach to declarative debugging (error diagnosis) of logic programs. The main contributions are:

• "Declarative source debugging", a new approach requiring less queries from the user.

• Improved algorithms for classical "declarative execution debugging".

• Support for Prolog impurities, such as cuts and side-effects.

• The use of suspect trees, allowing uniform treatment of all considered bug types.

• More extensive use of the available user knowledge.

• An approach for debugging meta-interpreted and pre-processed programs.

• An implementation architecture designed for large program computations, but not tested on large programs.

• An experimental prototype incorporating these improvements.

• An application of the debugging framework to the problem of knowledge base updating.

The concepts and methods are layered according to the programming language: from pure Horn logic (normal) programs to full Prolog.

Notation

We use $\log_b X$ to denote the logarithm of X for power base b, and simply log X for $\log_2 X$.

The main concepts and definitions are listed in the index at the end.

Table of Contents

Figures

Following is the list of figures. Please refer to the index at the end and lookup entries for *Figure* to find them.

Tables

Following is the list of tables. Please refer to the index at the end and lookup entries for *Table* to find them.

Preambule

This thesis has four parts. The first five chapters deal with a new theoretic framework for declarative Prolog debugging diagnosis. The next part describes a prototype implementation of the theory, a Prolog debugging environment. The third part relates the application of the debugging technology to the problem of knowledge base updating, as an example of possible cross-fertilizations with other research fields. There's finally a conclusion, summarizing the main achievements and open problems.

The reader is expected to have basic notions about Prolog and logic programming, say as contained in [17]. Some basic experience with a conventional Prolog tracer debugger would also help. The ideal reader will also have read chapters 1-3 of [83], a main reference for this work.

Here's a table for those in a hurry, pointing possible reading paths:

Who you are	Start at...	... and continue with
Prolog User	Given the experimental character of the implementation, your only motivation should be be to get an implementor look into this, so that later you may become a user; cf. "Environment implementor"	
Environment implementor	Chapter 6, which describes the HyperTracer concept prototype. It also provides an overview of the main technical aspects.	Chapter 7 analyses possibilities for an efficient implementation.
Theortician	Chapter 1 includes a section on the main results of the thesis, with pointers for other parts of the text.	Follow the most provocative claim in chapter 1.

Table 1: Reading paths

The present work originated several reports and publications during 1987-90, namely [67], [66], [68], [69], [13] and [71]. The associated implementation work provided the debugger prototypes for the ESPRIT ALPES project, as well as for a R&D contract with Apple Computer.

Part I: Theory

1. Context

We start with an abridged historical perspective of previous work on Prolog debugging. Following is a summary of needs, and a summary of the main accomplishments of this work regarding those needs.

The impatient reader may prefer to take a quick tour at chapter 6 before continuing, which describes our experimental prototype, and in particular to the section “Example debugging session”, which gives a flavour of a debugger based on this work.

1.1. Some history

Prolog started [80] with no support for debugging. The first widely used efficient Prolog implementation [73] had only a simple trace/notrace facility, tracing all Prolog predicate calls. Later Byrd [11] added to it the first Prolog specific tracer.

Byrd's main contribution was the introduction of the 4-port box model of execution. Whereas deterministic language procedure calls can be represented with 2-port boxes, for CALL and EXIT ports, Prolog needs 4:

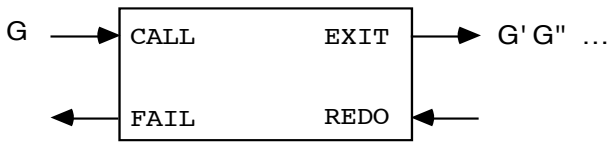

Figure 1.1: 4-Port Box

For each solution to goal G, execution flow goes through the EXIT port. On backtracking for more solutions the REDO port is used, and when no more solutions are available, execution for the predicate terminates through the FAIL port.

Byrd introduced this box model, and developed a tracer providing features alike those present in debuggers for other languages, but adapted to deal with the non-deterministic nature of Prolog execution: commands for “creeping” instead of “single-stepping”, “skip” instead of “jump over”, “spy-points” instead of “breakpoints”, and so on. Even today, most commercial Prolog tracers are based on these features.

Following Byrd's work many refinements were developed during the 1980s: additional features, more ergonomic interfaces, providing more detail, etc. Some of these will be overviewed in the last chapter.

The tracer approach satisfies the basic requisites of a Prolog debugging tool. But debugging practice did not deviate much from that of debugging programs in deterministic languages, with the added trouble of dealing with more complex execution traces, due to Prolog's nondeterminism. We might visualize a tracing session as follows, with the arrows denoting the user´s whereabouts within the execution trace:

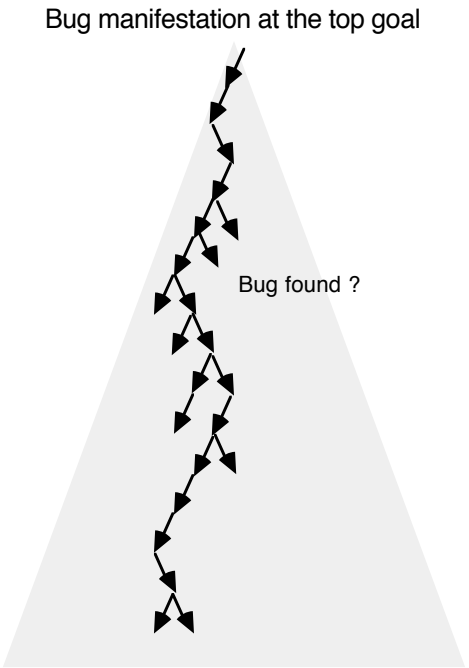

Figure 1.2: Tracing

In summary, *tracers do not capitalize on Prolog's specific features*, such as its clear declarative semantics, but *they mirror Prolog's idiosyncrasies*.

This situation was perceived by Ehud **Shapiro**. The first part of his landmark PhD thesis [83] introduced *Algorithmic* (or *Declarative*) bug diagnosis[1]. His basic idea was to use an oracle, an entity with knowledge about the intended declarative semantics of a program, namely its author, so that with declarative knowledge alone a debugger (or diagnoser) could pinpoint a bug in the program:

program + oracle knowledge ∅ bug instance

Whereas in tracers the user navigates about the execution trace using operational commands, entirely on his own initiative, with a declarative debugger he navigates according to the algorithmic criteria of the debugger, based on declarative knowledge expressed by the oracle. Because navigation is now based on declarative information, and conducted in a systematic and optimized way, a bug is guaranteed to be found, typically with less user interaction and effort.

In practical terms, a debugger implementing this approach might originate a debugging session as follows:

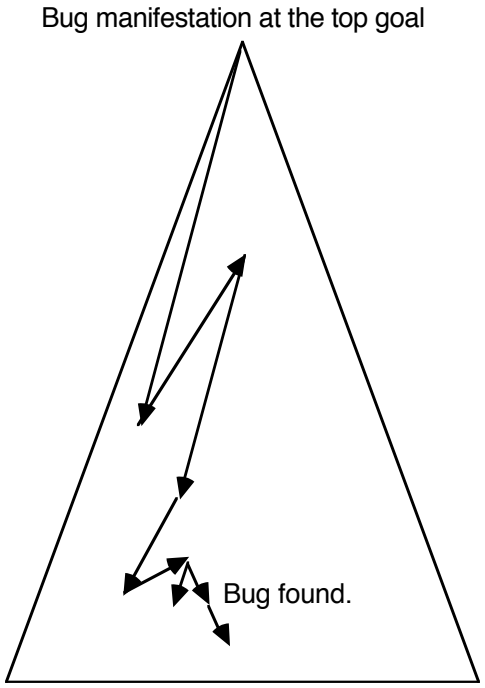

Figure 1.3: Algorithmic Debugging

[1]The second part deals with the bug correction problem, also an important contribution to the artificial intelligence learning field, but of less interest to us

The declarative debugging approach brought hope for better debugging tools, and originated subsequent research work, which we later overview (cf. chapter 9) and compare to our framework.

The present thesis continues this trend, extending the programming language acceptable for declarative debugging, introducing new and improved debugging algorithms, and coming up with a prototype implementation integrating also some tracer features.

1.2. Summary of field needs and thesis's main results

At the end (cf. "Conclusion" chapter) we'll overview work by other authors, positioning it in terms of our own framework. Following is a summary of problems and needs motivated by the previous work on declarative debugging, together with our own contributions to them:

• **Declarative debugging has focused exclusively on the search for bug instances in computation traces, rather than bugs in the program source**.

> *We present the first "declarative source debugging" algorithms, and show how they require less queries to find a diagnosis(cf. chapter 5).*

• The **algorithms used are suboptimal** regarding the number of queries for searching bug instances[1].

> *We present algorithms parametrized by the level of optimality desired, for all bug types (such as Generalized Divide&Query(N) - cf. chapter 2).*

• Existing diagnosis algorithms may require program computations different from the original one producing the bug manifestation, hence leading to **nontermination problems**.

---

[1] Some are optimal for binary or balanced suspect trees (as in [83]; others require a different type of oracle queries (equivalent to queries about nonatomic goals) to "transform" the suspect tree into a binary tree [76]. In both cases, only the wrong solution problem was considered.

Our algorithms work on a frozen ("post-mortem") representation of the computation trace, thus avoiding (debugger-originated) differing recomputations, and are guaranteed to terminate if the original computation did (cf. chapter 2, section "Finding Bug Instances", subsection "The basic algorithm").

• **Oracle information is not fully used**.

Our debugging framework uses subsumption among oracle statements for planning the next queries, and not just when the user is about to answer them (cf. subsections "Refining suspect sets with the oracle" and "The bug instance search tree", chapter 2)

• **Prolog's impure features** (namely cuts and side-effects) are at most tolerated but ignored, i.e. they're not specifically supported by declarative diagnosis algorithms.

We present the first declarative debugging methods for programs contaning cuts[1] *and output side-effects; we give a partial solution for the problem of programs using internal database side-effects (cf. chapter 4).*

• There are no methods available for finding **bugs in interpreted or preprocessed programs**.

We present methods for debugging such programs (cf. chapter 3).

• All authors use **different algorithms for different bug types**, rather than uniform polymorphic algorithms.

We use a uniform framework, based on the notions of suspect tree and suspect set. Our algorithms can be used for all bug types other than non-termination, by working with different suspect types (cf. Theory Summary section on chapter 6). This reflects into a uniform implementation (cf. chapter 7).

• Most implementation **designs are limited to toy programs**[2].

---

[1] Our basic approach to cuts, mentioned in [66] and implemented in a prototype demonstrated to the referees of the ALPES (ESPRIT P-973) project in September 1987, is in essence identical to that of [40], which was developed independently.

[2] With the exception of [76], but whose restrictions make it uninteresting in most settings (cf. previous footnote on the nonoptimality of previous algorithms).

We introduce an architecture based on "virtual trace storage", a mixed storage/recomputation technique to minimize space and time costs[1]*. However our experimental implementation was done at a high level, in Prolog, and so currently only toy programs are tackled by it (cf. chapter 7).*

• Existing debuggers have **modal interfaces**[2], forcing the user into a rigid mode of question/answer interaction.

Our prototype provides a user-friendly graphical interface, allowing a mixture of declarative debugging and execution browsing[3]*; several top goals can be examined at once, with different algorithms being applied arbitrarily and incrementaly, under user control (cf. chapter 6).*

• There are **no distributed declarative debugging systems**. Declarative debugging of concurrent languages is currently done via sequencial interpreters.

We didn't address this problem.

• Declarative debugging shares **similarities with other fields**, suggesting cross-fertilization.

We established a conceptual and practical bridge with the field of knowledge base updating (cf. chapter 8).

As a **global accomplishment**, we believe to *have managed to integrate all the contributions above in an elegant way*. Wrong solutions, incomplete solution sets, inadmissible goal calls and wrong output segments are all treated with the same diagnosis algorithms, for any logic programming language resorting to SLDNF operational semantics. Logic programming language impurities reflect simply into additional bug types or extended suspect sets. All this can be appreciated at a glance in the "Theory Summary" section, chapter 6.

---

[1] Our approach can be seen as a generalization of [76].

[2] As opposed to non-modal interfaces, where the user has more freedom to interact and to pick interface objects to act upon, as in standard "Graphical User Interfaces" (Apple Macintosh, Microsoft Windows, Motif, NeXTStep, etc.).

[3] This approach was presented in [66], as a design solution for the ALPES debugger. Another system which also advocates an hybrid style of interaction is the Transparent Prolog Machine [34], developed independently.

The "HyperTracer" prototype reflects this integrated and uniform character.

On the down side, it's only fair to refer the current unefficiency of the HyperTracer, which is due to an experimental implementation and not to its design, as discussed in the "Implementation Issues" chapter. We'd certainly like it to already be an usable tool, but unfortunatly the manpower for this work was finite. There's still work to be done after this thesis !

2. Basic framework

In this chapter we present the basic framework[1] to define and solve the following problem. Given:

• A consistent oracle theory, extensible on demand and interactively by the user.

• A normal logic program, with implicit program completion, and using the Negation As Failure rule [49].

• A finite and non-floundering SLDNF computation [ib.], implementing an exhaustive search for solutions, and producing an incorrect result.

Find:

• An incorrect program component instance, a diagnosis, without additional (different) program computations, and minimizing the cost of querying the user.

Essentially, the framework defined in this chapter improves the treatment for normal programs by other authors, and lays down the foundations for the next chapters. We come up with the notion of suspect tree, the cornerstone of our approach, and several new algorithms based on it.

---

[1] More extended than the original version presented in [68].

2.1. Definitions

In this section we'll define the theoretic concepts needed for declarative debugging. In addition to the usual concepts related to the buggy logic program, we also need a theory about the program and its computations, to be later implemented as a logic program, and on which the debugging algorithms will be based. To this we call the (meta-level) *debugger theory*, as opposed to the (object-level) *program*. We'll use the debugger theory concepts as part of our notation, and will associate condition statements in the text to truth in terms of the debugger theory.

An interesting point to note is that the knowledge expressed in the debugger theory and the debugger algorithms are cleanly separated, rather than amalgamated as other authors have in practice done for their debugger implementations. Conceptually the present approach corresponds essentially to viewing a declarative debugger as a "post-mortem" procedure, to be applied after a computation has terminated, rather than a runtime interpreter/debugger.

2.1.1. Programs

A *program* is a set of predicate definitions.

A *predicate definition* is a set of zero or more clauses, plus one predicate completion rule. All predicate symbols have a predicate definition. A *clause* is an implication of the form H <- B, where H is an atom, the *clause head*, and B is a conjuntion of positive and/or negative literals, the *clause body*.

Clauses with empty body are *facts*, and the others are *rules*. The *predicate completion rule* is syntactically implicit, and means "this predicate has no more clauses". Clauses and completion rules are *program components*, and have unique names. We can say either that a program is a set of predicate definitions, or a set of clauses plus a completion rule per predicate symbol.

For each clause H<-B in the program there's a correspondent fact clause(H,B,Ref) in the debugger theory, where Ref is the clause name. For each predicate definition for predicate symbol P in the program, or equivalently for each completion rule, there's a correspondent fact predicate_definition(P,Ref), where Ref is the completion rule name. Program clause variables are represented as unique constants in the debugging theory.

2.1.2. Computations

Program components are used to build SLDNF trees by an interpreter, at least conceptually. Most interpreters do so in a much more efficient way than the following definitions suggest. Typically they follow a strict search strategy, so that most of the SLDNF structure remains implicit.

A *goal* is a conjunction of positive and/or negative literals. An *atomic goal* is a single positive literal (or atom). A *clause instance* is a copy of a program clause with an applied substitution. For convenience, we'll assume that all clause instances have different names, even if they're syntactically similar.

The *SLDNF-tree* for a goal G and an implicit program, denoted by SLDNF(G), is defined as follows, based on [49] but with some changes, to represent explicitly the derivations for negated subgoals.

• Each node contains a goal, and a selected literal in it.

• The root node is G.

• Tree links can be of one of two types: *clause match links* or *subderivation links*. Clause match links are labelled with a substitution; subderivation links have no label.

• A node with the empty goal has no children.

• Let G1…Gi…Gn be a non-empty node and Gi, the selected literal, a positive literal or atom. Then the node has a child (G1 … Gi1 … Gim … Gn)θi for each instance of a program clause H<-Gi1…Gim, obtained by applying the substitution θi such that Gi and H are unifiable with mgu θi. Each such child node is linked to the parent node by a clause match link, labelled with <“name of H<-Gi1…Gim ”, θi>.

• Let G1…Gi…Gn be a non-empty goal node and Gi, the selected literal, a ground negative literal ~A. Then the node has a child, the root of SLDNF(A), connected by a subderivation link, and it may or not have another child:

  •• If SLDNF(A) has a solution path (cf. below), the node has no more children.

  •• Otherwise the node also has a child G1 … Gi-1 … Gi+1 … Gn, linked by a clause match link labelled with <“”, ε>, where “” signifies the absence of a clause name.

The main difference regarding the conventional definitions of SLDNF trees lies in the treatment of negative literals: failed subderivations *have* a representation in the tree, for debugging's sake as will become clear later.

Example Take the following program, producing top goal solution a:

```
a :- b, ~c.

b.    c :- ~d.    c:-e.       d.
```

The SLDNF-tree will look as follows:

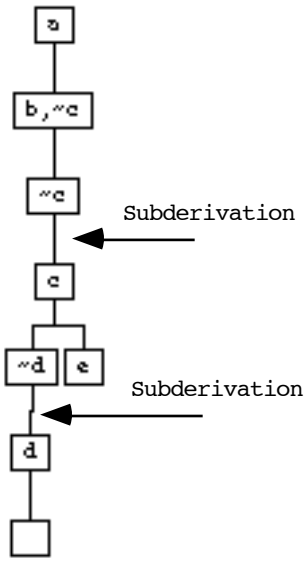

Figure 2.1: A SLDNF-Tree

→

A *goal call* is a selected atomic goal, in some node in a SLDNF-tree. The *goal call binding* is the composition of substitutions accumulated from the root down to the node of the SLDNF-tree where the goal call is. Notice that it can involve variables which are not present in the goal call literal.

We now define an important relation among nodes, “under”, which specifies the notion of composition of independent subcomputations. Consider a SLDNF-tree, and 2 goal calls T and G in it. G *is under* T iff either:

• G is a *computed instance*[1] of an atomic goal in the clause body of a clause matching T.

---

[1]An instance obtained via the SLDNF construction process above.

• G' is a computed instance of an atomic goal in the clause body of a clause matching T, and G is under G'.

• ~G' is a computed instance of a negative literal in the clause body of a clause matching T, and G is under G'.

Notice that if G is under T, then G is a descendent of T in the SLDNF-tree, but the opposite does not hold: in general T has descendents in the SLDNF-tree which are not *under* it.

Example Take the following program, and top goal a:

```
a:-b,c.     b:-e.     e.    c.
```

The SLDNF-tree is:

Figure 2.2: SLDNF-Tree illustrating *under*

Goal call `e` is *under* `b`, but `c` is not. →

Take a SLDNF-tree, and a node G1…Gi…Gn, with selected goal Gi. A *solution path* for goal Gi is a path of clause instance links in the tree, starting at the node and ending at the first node below it which does not contain goals under Gi (if such a node exists, otherwise there's no solution path)[1].

Let G be a goal call in some SLDNF-tree. A *goal solution* for G is the atom instance Gθ, where θ is the sequence of substitutions along a solution path for goal G. The *goal solution binding* for Gθ is the substitution accumulated from the root of the SLDNF-tree until the end of the solution path. Notice that it is at least as specific as θ: it is {goal call binding} ≈ θ.

---

[1] Notice that this definition abstracts from the execution order of goals. I.e. Prolog's left-to-right strategy is *not* assumed.

A *predicate instance* is a pair <G,P>, where P is a predicate definition for atomic goal G. It denotes the subset of clauses of P that match G.

A *predicate body instance* is the set of all goal calls in clause bodies of a predicate instance, resulting from computed instances of their literals. Sometimes we'll also use this concept regarding negative literals, meaning negative literals ~Bi in which Bi is in the predicate body instance.

Notice that a literal in a clause instance may originate multiple goal calls in a predicate body instance.

Example Consider the following program, and top goal a(a1).

```
a(a1):- b(X), c(X).     b(1).       b(2).

a(a2).
```

The predicate instance of predicate a/1 for goal a(a1) is set comprising the first clause. The predicate body instance for goal a(a1) is the set of goal calls {b(X), c(1), c(2)}.→

2.1.3. Goal behaviors

A *goal behavior* expresses the result of a program computation for an atomic goal, and is obtained from a SLDNF tree containing it. It consists in a goal call in the tree plus its solution set, and has a representation in the debugger theory, as a set of facts. For each atomic goal, the debugger theory contains the following facts:

• goal(GN,G,Cref,Pref), where G is an atomic goal with unique name GN, occurring in a parent clause named Cref and matching (or attempting to match) predicate instance named Pref; its (object-level) variables are existentially quantified;

• solution(GN,N,S,Cref), for each solution S, uniquely named N, obtained from the head of matching clause instance (with S's binding) named Cref, for goal call named GN; its (object-level) variables are universally quantified, and reflect the goal solution binding according to the definition in the previous section.

Given a goal behavior, a *goal behavior facet* (or *goal facet*) is some part of it - either a *solution* or the whole *solution set*, and in any case having a unique name[1] - and corresponds to a tuple or tuple set in the 'solution' relation above. We'll represent it by its name, and sometimes use a functional notation:

• call(G) denotes the name of goal G.

• solution(G) denotes the name of some solution for goal G.

• solution_set(G) denotes the name of the solution set for goal G.

• solution_set$_{\varnothing}$(G) denotes the name of the solution set for G, when it is empty; it's a particular case of the previous one, and just a notational convenience.

We say a program component instance CI *matches* a goal behavior facet F if:

• F is a solution, and CI the clause instance producing it (i.e., labelling the topmost link in the solution path), with the solution binding.

• F is a solution set for goal G, and CI is the predicate instance for G.

2.1.4.Core oracle theory

An *oracle* is an entity with knowledge about the *intended semantics* of the program: it characterizes the intended program (composed of its intended predicates) rather than the present program, regarding the correctness or incorrectness of specific program computation results. Independently of the type of semantics involved, its knowledge is made partially explicit as a relation of “correct”/“incorrect” assertions, and part of the debugger theory. The oracle has the ability to make it as explicit as required by a debugging algorithm, by adding tuples to the relation.

In addition to those assertions, the oracle may also contain assertions about the correctness of some program components, which encapsulate knowledge about the trusted parts of a program, such as system predicates for example. Such assertions form a relation correct_component(Program component name). We'll return to it in the next section, and for now concentrate just on “oracle statements”.

---

[1]Unique names can be easily provided by the interpreter implementation, for example by using <execution timestamp , process> pairs.

Oracle statements are tuples for the predicate relation o_s(Status,F). The first argument is either of 'correct' or 'incorrect', and the second is a goal behavior facet name. Each oracle statement classifies a goal facet as correct or incorrect, independently from other goals (i.e., the correctness of a goal behavior facet is strictly independent of that of other facets) and also of any operational aspect (e.g. regarding execution order among goals, or any other external factors). In other words, oracle statements are entirely context-free, and express solely the desired declarative semantics of the program.

Typically the oracle is based on *user statements* (cf. below) given by the author of the program, via declarative yes/no answers: he is presented with one or more goal behavior facets *within a complete goal behavior*, and is asked whether it is correct or not.

Following is a table with the possible cases for the goal behavior facet types, and their meaning, according to the intended program semantics known by the oracle.

Type	**Status**	**Meaning**
solution	correct	solution is correct (even though other solutions may be missing)
solution	incorrect	solution is incorrect (even though other solutions may be missing)
solution set	correct	all solutions for the goal are correct and none is missing
solution set	incorrect	the solutions produced for the goal are correct, but some are missing

Table 2.1: Basic oracle framework

Notice that a goal can have both an incorrect solution and a missing solution in its behavior. In such cases it is arbitrarily assumed that the existence of an incorrect solution is interesting for debugging, but the incorrectness (i.e., incompleteness) of the solution set is not; the motivation for this choice is the fact that incorrect solutions are easier to diagnose than incomplete solution sets, at least for programs without negation, as will be seen later (cf. definition of suspect trees below).

The *oracle theory*[1] is made of the oracle statement and user statement relations, plus some auxiliary formulas (cf. below). It has a generic *core* and a *customized* part, which depends on the logic programming dialect.

The oracle must always be consistent regarding the single[2] following *oracle theory integrity constraint*:

$$\neg (o\_s(incorrect,F) \wedge o\_s(correct,F)).$$

An *user statement* is represented as a tuple for the relation u_s(Status,F), and reflects into an oracle statement via the following clause:

$$u\_s(Status,F) \Rightarrow o\_s(Status,F)$$

We distinguish the two statement relations (o_s and u_s) to emphasize their different justifications (debugger and user knowledge, respectively).

Given the meaning associated to statements about correct solution sets, the following rule is also present - *any* statement about the completeness of the solution set implies the correctness of each individual solution:

$$o\_s(\_,solution\_set(G)) \wedge solution(G,S,\_,\_) \Rightarrow o\_s(correct, S).$$

This has just the practical consequence of forcing the human interface of the debugger to restrict some user statements.

If all solutions are correct and the solution set is not incomplete, the solution set is correct:

$$[solution(GN,S,\_,\_) \Rightarrow o\_s(correct,S)] \wedge \neg o\_s(incorrect, solution\_set(GN)) \Rightarrow o\_s(correct, solution\_set(GN)).$$

Oracle statements refer (unique) goal behavior facet names, and therefore apply to a single computational entity. Reuse of user statements can be achieved, when a goal facet *subsumes* another, using the following oracle rules:

$$o\_s(correct,F1) \wedge subsumed\_facet(F1,F2) \Rightarrow o\_s(correct,F2).$$

$$o\_s(incorrect, F1) \wedge subsumed\_facet(F2, F1) \Rightarrow o\_s(incorrect, F2).$$

---

[1]For the sake of clarity we'll use loose first order logic notation, with implicit universal quantifiers, and underscores representing universal “don't care” variables.

[2] We'll introduce another constraint after we define bug instances in the next sections.

subsumed_facet(F1,F2) holds iff F1 is at least as general as F2, in the sense that the two previous rules would be valid were user statements available about F1 and F2. This concept is akeen to logical subsumption, and it also embodies as a particular case the obvious idea of "remembering previous user answers" found in other declarative debuggers.

The subsumed_facet predicate depends on the semantics of the logic programming dialect, in particular of its level of logical "purity" (use of side-effects, etc.), and will be defined in the customized part of the oracle theory.

Example Take two goal solutions computed from the same program, p(1,X), p(1,a), the first of which is considered correct by the oracle. Assuming that the statement was made considering X with an implicit universal quantifier, as would be the case in a situation involving pure logic programs, then p(1,a) can be considered correct, even without asking the user. →

2.1.5. Customized Oracle Theory

In addition to the core oracle theory, additional rules must be considered for each specific logic programming language feature.

Given the negation by failure rule, the oracle theory also contains the following rules, in the right column of the next table. Each corresponds to the basic logic equality on the left column.

∀¬G ⇔ ¬∃G	o_s(correct,solution(~G))⇒o_s(correct,solution_set∅(G))
∃¬G ⇔ ¬∀G	o_s(incorrect,solution_set∅(~G))⇒o_s(incorrect,solution(G))
¬∀¬G ⇔ ∃G	o_s(incorrect,solution(~G))⇒o_s(incorrect,solution_set∅(G))
¬∃¬G ⇔ ∀G	o_s(correct,solution_set∅(~G))⇒o_s(correct,solution(G))

Table 2.2: Embedding Negation As Failure

For example, the first line reflects the fact that a negated goal solution (with an implicit universal quantifier) is valid iff the same (non-negated) goal solution literal finitely fails, given the negation as failure rule. We may therefore forego oracle statements about negated literals, and prompt the user only to obtain statements about atomic goals.

For normal programs, without the impurities found in most practical logical programs, it is possible to define a subsumption relation between goal solutions[1]:

subsumed_facet(solution(S1),solution(S2)) ← “S1 subsumes S2”.

S1 subsumes S2 iff, modulo variable renaming, S2 is equal to S1 or it is an instance of S1.

Again for normal programs, there's a particular case where identical facets provide some correctness information directly. If a goal has a solution subsuming itself (i.e., syntactically identical), then its solution set can't be incomplete: all possibly missing solutions are subsumed by the existing one. However it may (by subsumption) contain incorrect solutions.

goal(GN,_,_,_) ← solution(GN,SN,_,_) ← subsumed_facet(SN,GN) ⇒
¬ o_s(incorrect, solution_set(GN)).

Whenever this rule applies, it is not necessary to obtain a user statement about the completeness of a goal solution set in order to conclude that it is correct. Information about correctness of individual solutions is enough.

Finally, if a correct program specification is available it can be included in the oracle theory, as suggested by [31], and later implemented by [25]. This can be done in the form of additional oracle rules, conditioning the relation o_s.

A specification can also relate directly to program components, rather than to computation results; for example “the clauses for predicate p are correct, but those of predicates q and r may or not be so”. This knowledge is represented in the relation correct_component(PC) referred in the previous section, PC being a name of a program component known for sure to be correct. It should be stressed that this relation does not directly state anything about the correctness of computation results.

The correct_component relation introduces the need for an additional constraint on the oracle theory, which will be presented after defining bug instances in the next section.

---

[1]It seems pointless to define a subsumption relation for solution sets: there seems to be no correlation between subsumption among goal calls and subsumption among solutions in their solution sets, due to use of negation as failure.

2.1.6. Bugs and their relatives

A *bug manifestation* is a goal behavior facet for which there's an oracle statement classifying it as incorrect. A program is considered buggy iff it has a bug manifestation. Since we're assuming the computation to be finite and nonfloundering, the only possible bug manifestations are incorrect solutions and incorrect (incomplete) solution sets.

An incorrect solution is also called a *wrong solution*, and an incorrect solution set a *missing solution* - as a reference to the solution missing in the set.

Intuitively, a *bug instance* is an instance of a program component in the SLDNF tree, such that it produces a bug manifestation independently of other bug manifestations. We'll define the two cases to be considered: wrong clause instance and incomplete predicate instance.

A *wrong clause instance* expresses the notion of buggy clause. Let H be a goal solution, matching clause instance H<-B1…Bn. This is a wrong clause instance if: o_s(incorrect, solution(H)) is true, and for each literal Bi in the body , o_s(correct,solution(Bi)) is true.

An *incomplete predicate instance* expresses the notion of buggy predicate completion rule, or of a predicate whose clauses fail to produce an additional solution, without any suspicion being assigned to their subgoals. It is a predicate instance <G,P>, for which o_s(incorrect, solution_set(G)), and such that for all goal calls in its predicate body instance, named Gi:

• if Gi is an atomic goal, then o_s(correct,solution_set(Gi)) is true

• if ~Gi is a negative literal with empty solution set, then o_s(correct, solution_set∅(~Gi)) is true

• (the case for successful negative literals is irrelevant)

The need to refer to correctness of whole subgoal behaviors, by referring to the *correctness* of their solution sets, and not just to their completeness, comes from those cases where wrong solution bindings cause a legitimate failure in a brother goal, but an incorrect failure in the father goal. This is taken care of in the first case above, via the core oracle theory's definition of “correct solution set” statements.

Example Take the following program, and top goal foo(1,Y), with an (incomplete) empty solution set.

```
foo(X,Y) :- q(Y), p(X,Y).
p(1,a).
q(b). % wrong clause
```

Consider that q(b) is incorrect, p(1,a) is correct, and goal p(1,b) has a complete (although empty) solution set. The guilt for the missing top goal solution should therefore be attributed to clause q(b), and *not* to the predicate definition foo/2. →

The ommission of the check of correctness for succeeded negative literals, as specified in the last case in the definition above, is acceptable because negation as failure produces no bindings, and therefore it is irrelevant to check if its success is correct: were it incorrect, it would not be the cause for the missing solution in the predicate.

Example Consider the program

```
foo(X,Y) :- ~a(X),b(Y).
```

Assume that top goal `foo` fails erroneously, but that `~a(X)` succeeded with a wrong solution, a(X) having failed with a missing solution. But if goal `a(X)` succeeded, `foo` would continue to fail; and it is not possible to change the binding produced by `~a(X)`, because it is empty. →Finally, a *bug* is a program component with a bug instance: a wrong clause instance is an instance of a *wrong clause*, and an incomplete predicate instance is an instance of an *incomplete predicate*.

2.1.7. An additional oracle constraint

Given the above definitions of bug, and the fact that the oracle may contain a relation correct_component(PC), stating that some program components are guaranteed to be correct, it's important to ensure the compatibility between both, with an additional oracle constraint simply formulated as follows: *A program component declared correct can not be a bug*. Or *any computed instance of a program component declared correct can not be a bug instance*.

This constraint could be stated more formally, by caracterizing the refutation of the definition of bug instance, in terms of oracle statements. Intuitively, it simply means that if the head of a correct program component instance is incorrect there must be an incorrect body literal, and if all body literals are correct the head must be correct.

Since the debugging methods below cannot violate it, because they don't consider correct program components as suspects, its motivation is just to serve as conceptual ground for the practical “inconsistent oracle” situation discussed in the last section.

2.2. Suspect trees, sets and their properties

We'll now prepare to follow the traditional diagnosis approach: start with a set of suspects, and eventually end up with a singleton, guaranteed to be a bug instance.

2.2.1. Definition

The *suspect tree* for a goal behavior facet F, ST(F), is based on the SLDNF-tree of the computation producing F, which is not necessarily at its root, and is recursively defined as follows:

• Each node has the name of a goal behavior facet, and is labeled with a program component instance

• The root is the node F

• Each *solution* facet node S is labeled with the clause instance matching it and, for each literal Bi in the clause instance body, it has a child as follows:

  •• if Bi is an atom, ST(solution(Bi))

  •• if Bi is a negative literal ~G, ST(solution_set∅(G))

• Each *solution set* facet node is labeled with the predicate instance matching it (cf. definition of "match" above in section "Goal Behaviors"), and, for each literal Bi in its predicate instance body, i.e. all goal calls immediately *under* it (cf. definition of "under" above in section "Computations"), it has one or more children as follows:

  •• if Bi is an atom, a child ST(solution_set(Bi))

  •• if Bi is a failed negative literal ~G, a child ST(solution(G')) for each solution G' of G

Example Consider the following program, with missing solution subset([2],[1,2]), taken from [49], page 127.

```
subset(X,Y) :- ~ (f_member(Z,X), ~ f_member(Z,Y) ).
f_member(X,[X|Y]).
f_member(X,[Y|Z]) :- f_member(Y/*bug*/,Z).
```

The suspect tree ST(failure(subset([2],[1,2]))) follows, shown here without the labels for the program component instances:

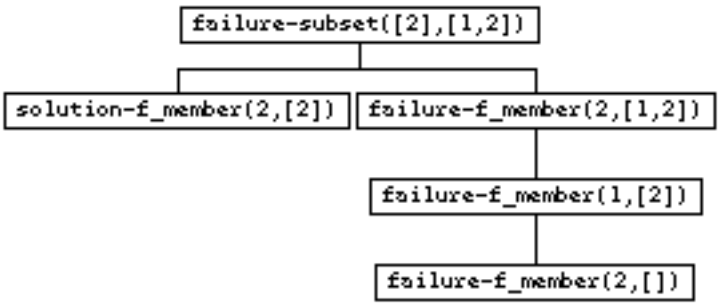

Figure 2.3: A suspect tree

→

The *suspect set* for a goal behavior facet F, SS(F), is the set of all program component instances labelling nodes in ST(F). Notice that similar component instances (say, equal clause variants modulo variable renaming) do *not* merge into a single suspect.

Finally, notice that these definitions apply to any goal behavior facet, and not necessarily to bug manifestations: considering or not those as such is a matter for the oracle to decide.

2.2.2. Existence of a bug instance in a suspect set

We now present a fundamental proposition.

Proposition 1 Given a bug manifestation T, its suspect set SS(T) contains a bug instance.

Proof SS(T) contains a bug instance iff ST(T) contains a node labelled with a bug instance. Now if T is a bug manifestation, o_s(incorrect,T) is true. Therefore there's a node D in ST(T) such that o_s(incorrect,D), and, for each of its children Di (if any), o_s(correct,Di); either T is such a node or, recursively, there's one in its children subtrees, given that ST(T) is a finite tree.

The result follows directly from the definitions for suspect tree and bug instance, that guarantee node D to be labelled with a bug instance: if D is a solution facet matching clause instance C, by mapping each of the node's children into a subgoal of C; if D is a failure facet, for a goal with predicate instance body P, by mapping each child into a goal in P. →

The proposition unifies potentially separate results concerning wrong and missing solutions. This is possible because we're abstracting from the bug manifestation types, through the use of generic goal behavior facets.

2.2.3. Refining suspect sets with the oracle

We'll now see how to make a suspect set smaller (*refine* it) by using oracle information, while guaranteeing that it contains a bug instance.

Proposition 2 Take a bug manifestation F and an oracle theory OT, and a node F' in ST(F) with program component instance PCI, a computed instance of a program component PC for which correct_component(PC) is true; then the suspect set SS(F)\{PCI} contains a bug instance.

Proof Similar to proposition 1, considering the tree obtained from ST(F) by removing the F' node, and making its children children of its father.→

A node matching a program component instance known to be correct is ignored, although its descendents are not.

Proposition 3 Take a bug manifestation F and an oracle theory OT, and a node F' in ST(F) for which o_s(correct,F') is true in OT; then the suspect set obtained from SS(F) by removing all suspects in SS(F') contains a bug instance.

Proof Similar to proposition 1, considering F' as a leaf of the original suspect tree ST(F).→

In other words, if the root of a suspect subtree is correct we can discard the subtree because there's a bug instance elsewhere. The dual case corresponds to using incorrectness statements, applying proposition 1:

Proposition 4 Given a bug manifestation F and an oracle theory OT, and a node F' in ST(F) for which o_s(incorrect,F') is true in OT, then SS(F') contains a bug instance.

Proof Follows directly from the definition of bug manifestation and proposition 1.→

We can thus be sure that, given enough consistent oracle statements, at most one for each goal behavior facet in the suspect tree, a bug instance will be found in any suspect set for a bug manifestation, since it is finite. Discarded suspects may also be bug instances, but at least one is guaranteed to remain (although not necessarily the one responsible for the original bug manifestation in the top goal).

For later convenience we'll now define notation for the outcome of refining. Given an oracle theory OT, and a bug manifestation F:

A *refined suspect set*, RSS(F,OT), is a minimal[1] set obtained from SS(F) by refining it using propositions 2, 3 and/or 4, using the oracle theory OT.

It is not necessarily unique, because of the oracle theory rules for subsumed statements, which may cause additional incorrect facets (i.e., roots for suspect subtrees) to be pointed out, allowing multiple uses of proposition 4.

Example Consider the following program fragment, and a top goal being executed by an interpreter that uses clauses in arbitrary order. Assume that during execution 2 goal calls are made in different parts of the top goal execution: even(s(X)), whose subderivation was constructed by the interpreter choosing the clause (1) first, and even(s(s(s(X)))), during which subderivation clause (2) was preferred first.

```
(a) even(0).  % bug: should be zero instead of a free variable
(b) even(s(s(X))) :- even(X).
```

Here are the 2 suspect trees for the subderivations:

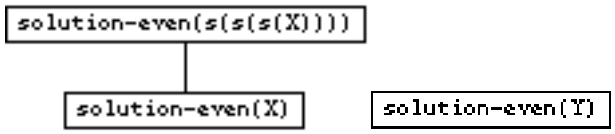

Figure 2.4: Two suspect trees for one statement

Now assume that the oracle stated o_s(incorrect, solution(even(s(s(s(X)))))). By knowing that even(s(X)) subsumes even(s(s(s(X)))), it follows that o_s(incorrect, solution(even(s(X)))) is true in the debugger theory. Therefore the suspect set {even(s(Y)):-true} could be focused on as the refined suspect set, instead of {even(s(s(s(X)))) :- even(s(X)), even(s(X)):-true}, with obvious advantage because it is smaller. →

A *refined suspect tree*, RST(F,OT), is the suspect tree associated to a particular RSS(F,OT). It is obtained from ST(F) by excluding the nodes whose program component instances are not in RSS(F,OT). It is not necessarily unique. Furthermore, if a node B under another node A in ST(F) has children nodes, and it is excluded in RST(F,OT), because it matches a program component known to be correct, then these nodes become children of A in RT(F,OT).

---

[1] Regarding inclusion, not cardinality. There may be multiple RSSs with different cardinalities.

Proposition 5 Given a bug manifestation F and an oracle theory OT, RSS(F,OT) contains a bug instance, and RST(F,OT) contains a node labelled with a bug instance.

Proof Follows directly from proposition 1 and the previous definitions.→

Lemma NON_EMPTY_REFINED Given a bug manifestation F and an oracle theory OT, RSS(F,OT) is nonempty.

Proof Follows directly from the previous proposition. →

2.3. Finding bug instances

2.3.1. The "game" of bug hiding

In order to develop diagnosis algorithms it's useful to view the diagnosis process as a two-player game, played between debugger and bug, the "moves" of the latter being represented by the oracle answers. Each debugger move consists in selecting a suspect to query the oracle about. Each bug move consists in an answer (correct, incorrect) by the oracle to a debugger query. The game begins with a given bug manifestation, and terminates when a bug instance is found. The debugger must imagine the worst case, i.e. that the bug is such as to maximize the number of moves needed to find it, whereas the debugger tries to minimize that number.

Of course, bugs are just sitting there, and they have no malevolous nature… But this analogy helps to clarify the debugger strategy: to choose the next "best" oracle query in this game.

2.3.2. The bug instance search tree

We'll now define the search tree for the "game of bug hiding". Its main motivation is to clarify the options available for the diagnosis algorithms.

But first let the *statement addition OT+OS to an oracle theory* OT of a statement OS = o_s(Status,F) be as follows: if OT|=OS, OT+OS is OT, else OT+OS is OT ≈ {u_s(Status,F)}. This operation is undefined if either OT or OT ≈ {u_s(Status,F)} is inconsistent.

The *bug instance search tree* for a bug manifestation T, given an oracle theory OT (possibly empty except for the integrity constraints and custom rules; i.e. initially there may be no statement additions), is denoted by BIST(T,OT), and is defined as follows:

• The tree has two types of nodes: *debugger nodes* and *bug nodes*.

• The root of the tree is the debugger node bist(T,OT).

• Each debugger node is labeled with a bug manifestation and an oracle theory, and is represented by bist(F,OT'). It has a (bug node) child query(Fi,OT') for each goal behavior facet Fi (except F) matching a suspect in the union of all RSS(F,OT') sets; it has no children if any of all RSS(F,OT') sets is a singleton.

• Each bug node is labeled with a goal behavior facet and an oracle theory; it is represented as query(F,OT'). Let its father be a debugger node with bug manifestation G. The bug node has the following children:

•• a (debugger node) child bist(G, OT'+{ u_s(correct,F) })

•• a (debugger node) child bist(F, OT'+{ u_s(incorrect,F) })

Notice that OT' doesn't imply u_s(correct,F) nor u_s(correct,F): if that were the case, this query node wouldn't belong to the tree.

Oracle theories in debugger nodes get progressively larger near the tree leaves, corresponding to more and more "possible user statements"[1] u_s(S,F); a larger oracle theory "at the start" (i.e., at the root) will tend to make the tree smaller.

Example Consider the following buggy program, producing wrong solution p:

```
p:-q.        q:-r.        r:-t.        t.
```

Here's the suspect tree for p, ST(p), where all nodes are solutions:

Figure 2.5: A suspect tree

And now the bug instance search tree, BIST(p,{o_s(incorrect,solution(p))}):

[1] I.e., statements hypothetically considered by the algorithm, but not yet pronounced on by the user.

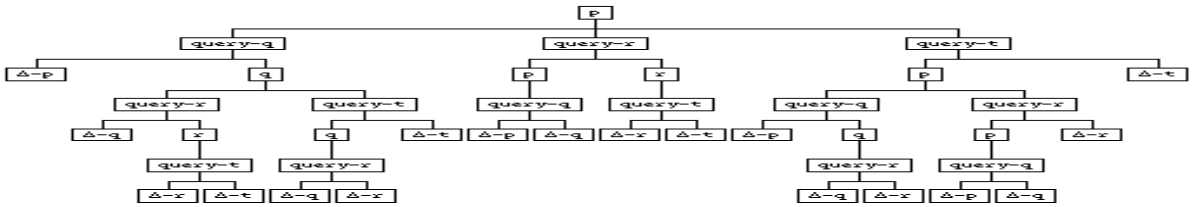

Figure 2.6: A Bug Instance Search Tree

Nodes above have the following meaning:

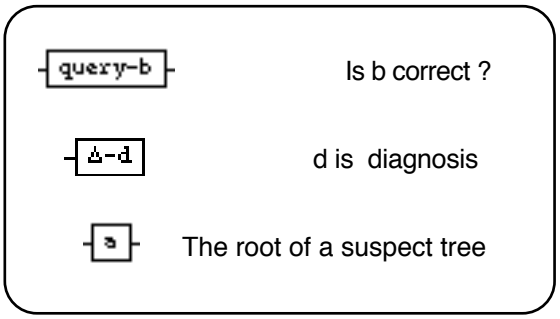

Figure 2.7: BIST legend

→

This tree makes explicit all possibilities of querying the oracle. We'll now make sure that we can actually find bugs using it! Notice in the previous example that diagnoses can be found at BIST leaves.

Proposition 6 Let T be a bug manifestation and OT an oracle theory. A program component instance is a bug instance if it is contained in a RSS(L,OT'), bist(L,OT') being a leaf of BIST(T,OT).

Proof All leaves are debugger nodes, because bug nodes always have two children. If a debugger node bist(F,OT') is a leaf, then at least one of the sets RSS(F,OT') has a single element I. Since by definition of debugger node F is a bug manifestation, then by proposition 5 I is a bug instance.→

Proposition 7 Given a bug manifestation T and an oracle theory OT, there are no certified bug instances matching goal behavior facets in non-leave nodes of BIST(T,OT).

Proof Take a debugger node bist(F,OT') with descendents. F being a bug manifestation by definition of debugger node, it may match a bug instance. But all sets RSS(F,OT') have more than one element, and therefore it may be the case that F doesn't match a bug instance.→

2.3.3. The basic algorithm

From the notion of bug instance search tree one can immediately define a straightforward diagnosis algorithm, which starts at its root and goes down to find a diagnosis at a leave, querying the user along the way to obtain user statements:

1. Given a bug manifestation T and oracle theory OT, compute BIST(T,OT). Make the root the current node.

2. If the current node is a leaf bist(G,OT'), return as diagnosis the single element of any RSS(G,OT') which is a singleton.

3. Select some child of the current node, a bug node query(Fi,OTi).

4. Query the user about the correctness status S of goal behavior facet Fi (“correct or incorrect ?”) and continue at step 2 with bist(Fi, OT ≈ u_s(S,Fi)) as the current node.

Notice that the user query at step 4 is always “necessary”, in the sense that its outcome is not yet known to the oracle. If that was the case, the corresponding query node in the BIST would not exist, because it would have been eliminated by previous suspect set refinement (cf. definition of BIST in the previous section).

It's also interesting to remark that the algorithm may produce as final result one of several diagnosis, whenever multiple RSS sets exist at the final step. For this to happen, the oracle must be able to “recognize” subsumed incorrectness statements. (cf. section “Refining suspect sets with the oracle” in chapter 2).

This algorithm should not look very “intelligent” to an user, because it may ask more queries than necessary, since no attempt is made at minimizing queries. But it serves as a conceptual basis for the improved diagnosis algorithms to be developed further on.

Notice that it is guaranteed to terminate, because each iteration makes the “current bug instance search tree” smaller, and the first tree is finite: this algorithm is defined over a “post-mortem” representation of a (finite) SLDNF tree.

2.3.4. “Intelligent” algorithm

An “intelligent” algorithm should find a diagnosis requiring as little information as possible from the oracle. For now we're solely concerned with “intelligence” as perceived by the user, abstracting from the computational cost involved.

A *query set* expresses the notion of a set of (oracle) user statements used to refine a suspect set, and is defined for a bug instance search tree node, as follows. Given some initial bug manifestation T and oracle theory OT, the query set for a debugger node bist(F,OT') in BIST(T,OT) is {all user statements in OT'}\{all user statements in OT}. A *diagnosis query set* is a query set for a leaf node in BIST(T,OT).

A *query sequence* is a particular ordering of a query set, corresponding to a path of bug nodes in a bug instance search tree. A *diagnosis query sequence* is a query sequence for a leaf node.

The *query sequence cost* for a query sequence is the cost, for the user, of producing the user statements in sequence. For now we'll assume this cost to be *the number of user statements* in the sequence[1].

We can now transform the basic algorithm into an "*intelligent*" one from the point of view of the user, by redefining step 3:

> 3'. Select a child of the current node, a bug node query(Fi,OTi), such that the *maximum cost* for any diagnosis query sequence starting in the node is minimum.

This algorithm is optimal, in the sense that it minimizes the maximum number of user statements needed to find a diagnosis - i.e. it concerns itself with the "*worst* case". Remark that for any query about goal behavior facet F, we do *not* have additional information or expectation on whether F is correct or not.

2.3.5. "Better" algorithms

Bug instance search tree are enormous[2], and therefore the previous "intelligent algorithms", as they were stated, lack practical interest for debugging non-toy logic programs.

Example Following is an example suspect tree for a goal behavior facet, omitting everything but node facet names:

---

[1] We'll return to this point in section "Improving the present framework", chapter 9.

[2] In the worst case, for a "flat" suspect tree with N nodes, the respective BIST will have N-1 *levels*.

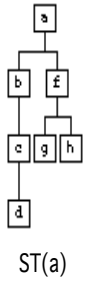

Figure 2.8: A suspect tree

And the corresponding bug instance search tree, showing just a small part in detail, a subtree rooted in one of the most promising query nodes, because the subtree height is smaller as can be gleaned from the larger picture:

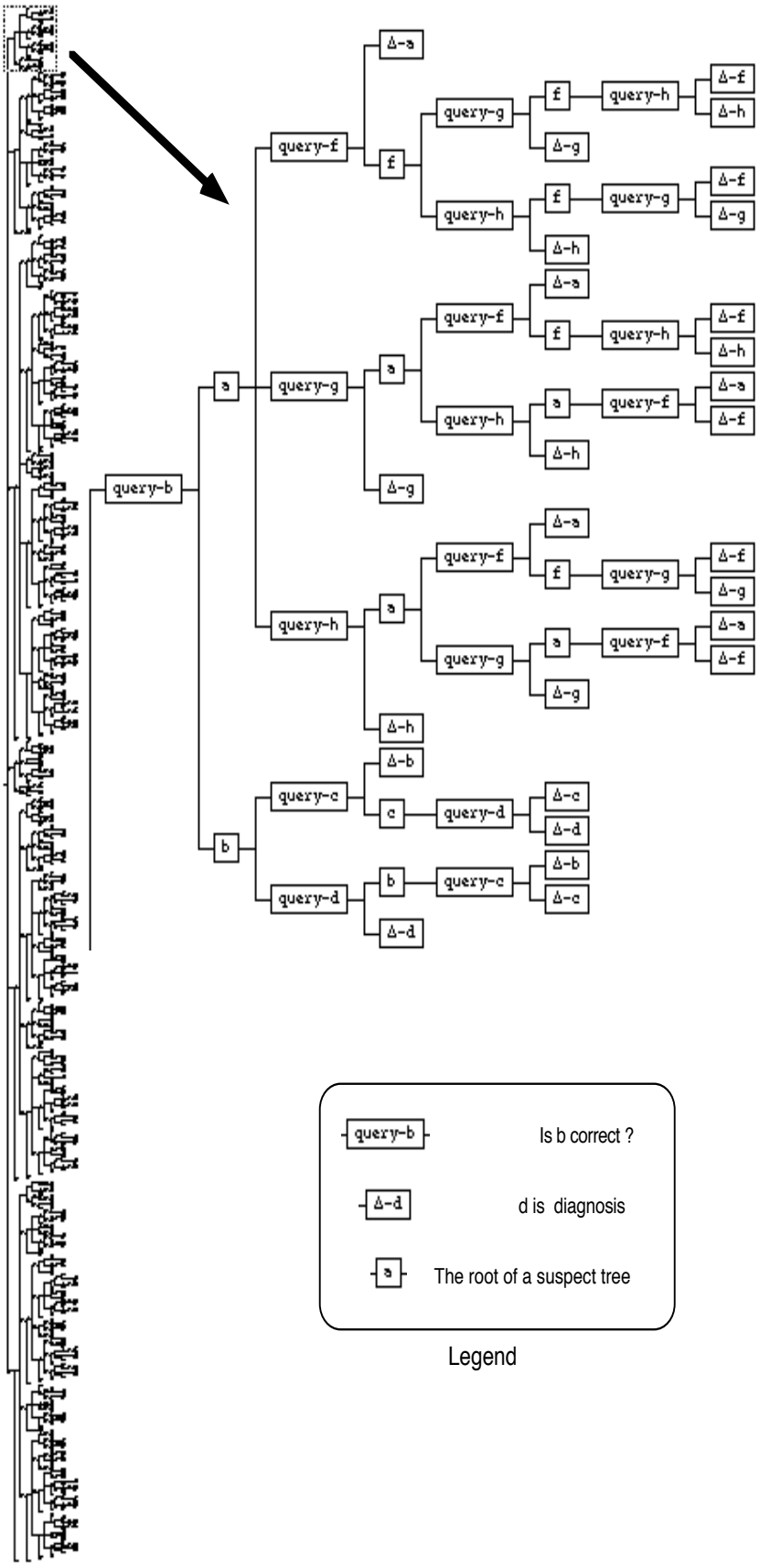

Figure 2.9: A "small" BIST

→

The purpose of the previous sections was to lay conceptual ground, rather than to provide an implementation scheme. As the past history of Artificial Intelligence has shown, we can improve the situation by using additional information, and thereby specializing the generic algorithms.

2.4. Search heuristics

2.4.1. Divide-and-Query: reconstruction and generalization

We'll now reconstruct, abstract and generalize Shapiro's Divide-and-Query (D&Q) algorithm, by using the previous algorithm with an "evaluation function" or heuristic that gives an upper bound on diagnosis cost.

Given an oracle theory OT and a bug manifestation F, a *smallest refined suspect set*, $RSS_{SMALL}(F,OT)$, is any of the RSS(F,OT) sets with smallest cardinality.

Proposition 8 Let T be a bug manifestation, OT an oracle theory, and bist(T,OT) the root node of BIST(T,OT). Consider any of $RSS_{MIN}(T,OT)$. It is possible to find a bug instance with a diagnosis query cost less or equal to its number of elements, $\#RSS_{MIN}(T,OT)$.

Proof In order to find a diagnosis with cost no greater than $\#RSS_{MIN}(T,OT)$, simply follow a path from the root to a leave, always passing through debugger nodes whose goal facets match elements of $RSS_{MIN}(T,OT)$. Each query node in the path "refines RSS_{MIN}", "removing" at least one of its elements.→

We now define the *Abstract Divide and Query* or *AD&Q* algorithm, so called because it will not be defined and used solely for the wrong solution problem, as other authors do, but instead abstracts from the type of suspects involved. It's obtained from the intelligent algorithm by redefining step 3:

> 3". Select a child of the current (debugger) node, a bug node query(Fi,OTi) with two debugger node children bist(A,OA) and bist(B,OB), such that $\max\{\#RSS_{MIN}(A,OA), \#RSS_{MIN}(B,OB)\}$ is minimum.

Notice that if each refined suspect set is unique[1], the generic upper bound above ($\#RSS_{MIN}$) is simply *the number of children of a debugger node*, or the current number of suspects. In other words, it particularizes to Shapiro's (suspect) tree weight.

Example Following is a buggy program example from [48], with a missing solution qsort_l([3,1,2],[1,2,3]):

```
qsort_l([],[]).
qsort_l([A|B],[C|D]) :-
    partition_l(A,B,L1,L2),
    qsort_l(L2,S2),
    qsort_l(L1,S1),
    qappend_l(S1,[A|S2],[C|D]).

partition_l(A,[],[],[]).
partition_l(A,[B|C],[B|D],E) :- A>=B, partition_l(A,C,D,E).
partition_l(A,[B|C],D,E):-  % should be partition_l(A,[B|C],D,[B|E])
    A<B,
    partition_l(A,C,D,E).

qappend_l([],L,L).
qappend_l([X|L1],L2,[X|L3]) :- qappend_l(L1,L2,L3).
```

The suspect tree for the missing solution above is:

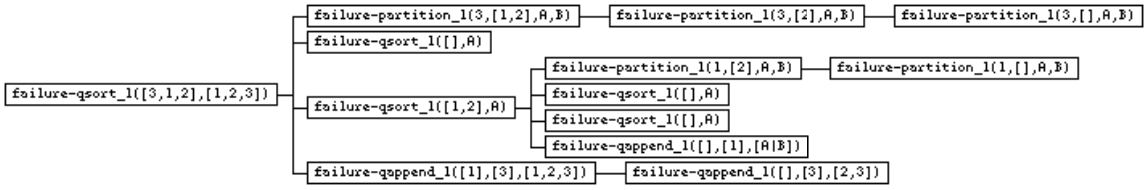

Figure 2.10: Suspect tree for a missing solution

And the query sequence for AD&Q follows, starting with the top goal:

[1] If no identical or subsumed statements are recognized.

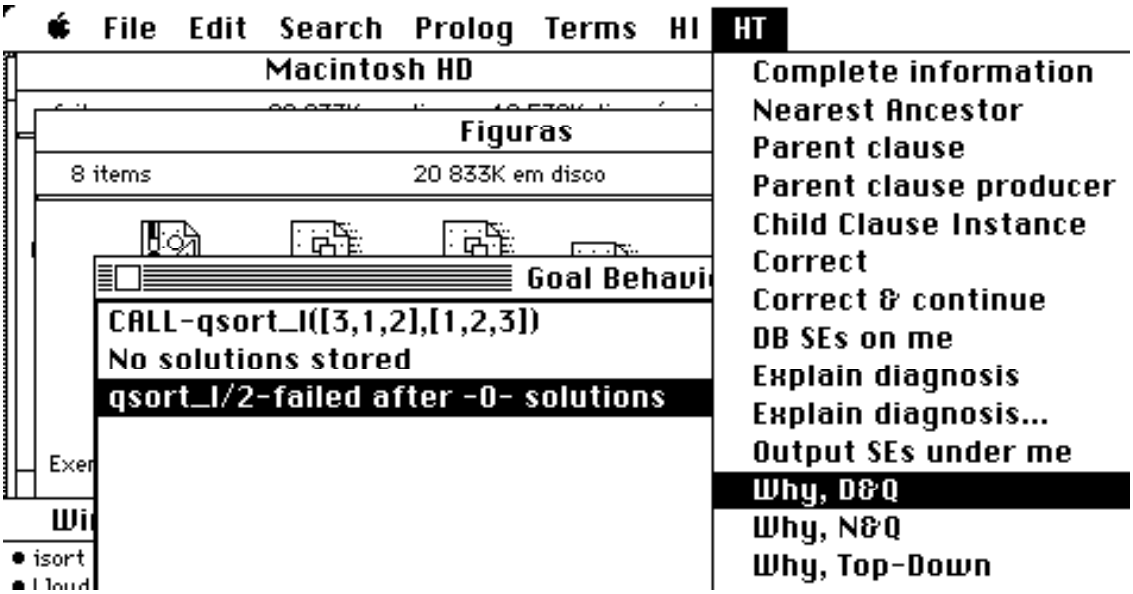

Figure 2.11: Complaining about a missing solution

Here's the full sequence:

Goal behavior facet term	Type	Correct ?	Comments
qsort_l([3,1,2],[1,2,3])	failure	no	top goal solution
qsort_l([1,2],[1])	solution	no	The user is shown the set of solutions for goal call qsort_l([1,2],L).
partition_l(1,[2],[],[])	solution	no	
partition_l(1,[],[],[])	solution	yes	Diagnosis

Table 2.3: A query/answer sequence

Only a total of 4 queries is necessary. [57]'s algorithm N.3, the best peforming in his comparison, needs 5. Our queries are identical, except for the query about goal call q_append_l([1],[3],[1,2,3]), which AD&Q skips because of its “divide and conquer” criterium.

After which the diagnosis is, as expected:

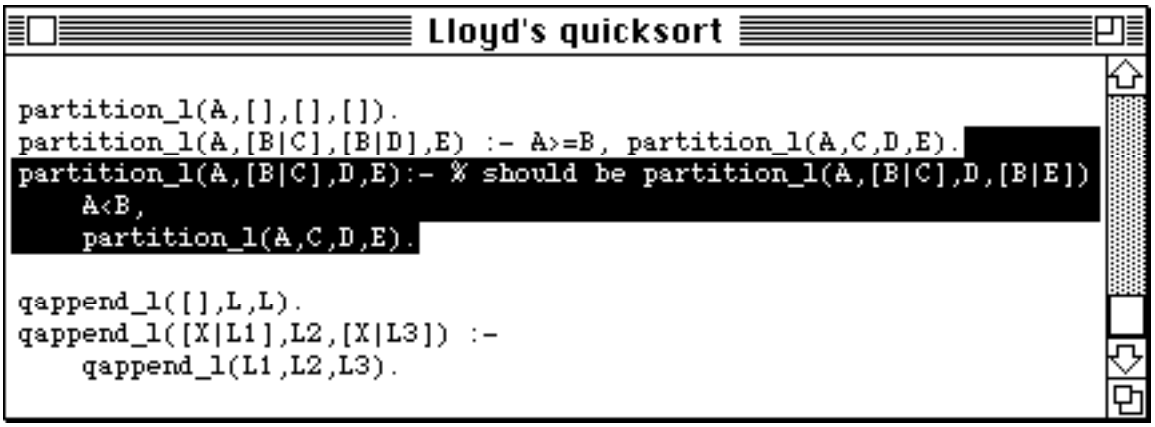

Figure 2.12: An HyperTracer diagnosis

→

AD&Q maximizes the suspect set refinement in a single query step, when we lack additional information about the probablity of the user statement being of the "correct" or "incorrect" kind. But in general it does not minimize the final diagnosis query sequence cost, because the intelligent algorithm's search is restricted to a "one-ply" game lookahead.

Example Consider the following suspect tree, and the oracle theory {o_s(incorrect,a)}.

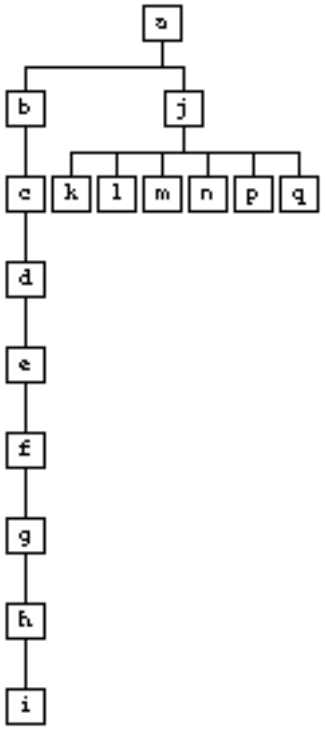

Figure 2.13: Unbalanced suspect tree

AD&Q would pick node b for querying next, because it is the "heaviest" node. However, the most malevolous bug instance could be matching one of the nodes on the right subtree. Therefore, in the worst case 8 queries would be needed: b,j,k,l,m,n,p,q, for example. But if our debugger decided to query about j first, in the worst case only 7 queries would be needed: j,k..q; if the bug instance was matching a node in the left subtree, less queries would be necessary, for example j,e,g,h,i. →

We now parametrize AD&Q with the number N of "game plies" to look ahead, obtaining a preliminary version of *Generalized Divide and Query(N)*, or *GD&Q(N)*:

> 3'''. Consider the set S_N of query node descendents of the current node that are N plies (2N-1 levels) below in the tree, or higher if they're leaves. Select the child of the current node, a bug node Q, minimizing $\max\{\#RSS_{MIN}(A,OA), \#RSS_{MIN}(B,OB)\}$; bist(A,OA) and bist(B,OB) being children of a node in S_N descending from Q.

GD&Q(1) is equivalent to AD&Q, and GD&Q(∞) to the intelligent algorithm, assuming query sequence cost to be the number of user statements.

2.4.2. Suspect tree form factors

We now turn to a more informed upper bound on diagnosis cost, to use as evaluation function, which takes into account the form of the suspect tree from which the bug instance search tree is defined. Recall, from the proof of proposition 8, that the previous upper bound was simply $\#RSS_{MIN}$, abstracting from the structure of the suspect tree.

We borrow a result from Shapiro, the upper bound on the number of queries for his D&Q algorithm[1]: $b \log_b n$, where b is the maximum number of children of any node in the AND tree of a wrong solution, and n the number of tree nodes.

Given a bug manifestation T and an oracle theory OT, the *form-aware bound on diagnosis cost*, $h_f(T,OT)$, is defined as follows: the smallest value $b \log_b n$ (for all refined suspect trees RST(T,OT)), where b is the maximum number of children of any node in RST(T,OT), and n is the number of nodes in the tree.

The final version of *Generalized Divide and Query(N)* follows:

---

[1] His result was actually b log n, which in general is too large; $b \log_b n = (b/\log b) \log n$ is a smaller bound, which also satisfies his recurrence relation I(n) ([83], page 47). By log X we mean $\log_2 X$.

3''''. Consider the set S_N of debugger node descendents bist(F,OT) of the current node, that are N plies (2N levels) below in the tree, or higher up if they're leaves. Select a child of the current node, a bug node Q, minimizing the maximum h_f(F,OT) for any descendent of Q in S_N.

Example Getting back to our previous example, it can be easily seen that GD&Q(1) will pick node j first, minimizing the maximum necessary number of queries until diagnosis. →

2.4.3. Complexity issues

The Abstract Divide & Query algorithm needs at most b $\log_b$ n queries, if each refined suspect set is unique[1]: it becomes essentially Shapiro's Divide and Query but abstracted from bug type.

The distinctness of Generalized Divide & Query(1) lies in its evaluation function b $\log_b$ n, which in practical terms makes GD&Q(1) query first about roots of wide (large b) subtrees; the resulting refined suspect tree may eventually have a smaller b, after the ellimination of its widest subtree. This situation is where GD&Q(1) manifests its advantages over D&Q. In general, the number of queries may be as low as b $\log_b$ n - [b + 1 - $\log_2$ (b+1)]. So *b+1-$\log_2$ (b+1) is the maximum gain to expect from the use of GD&Q(1).*

Example The following suspect tree, with b=6, will be "visited" by AD&Q from right to left in the worst case (AD&Q defines no ordering for subtrees with the same node count), whereas GD&Q(1) will visit it from left to right. The most malevolent bug will require 12 queries with AD&Q, instead of just 8 for GD&Q(1).

---

[1] If no subsumed statements are recognized. Otherwise additional oracle refinements will occur automatically, and less queries will be needed, which would correspond to the refined (removed) suspect subsets.

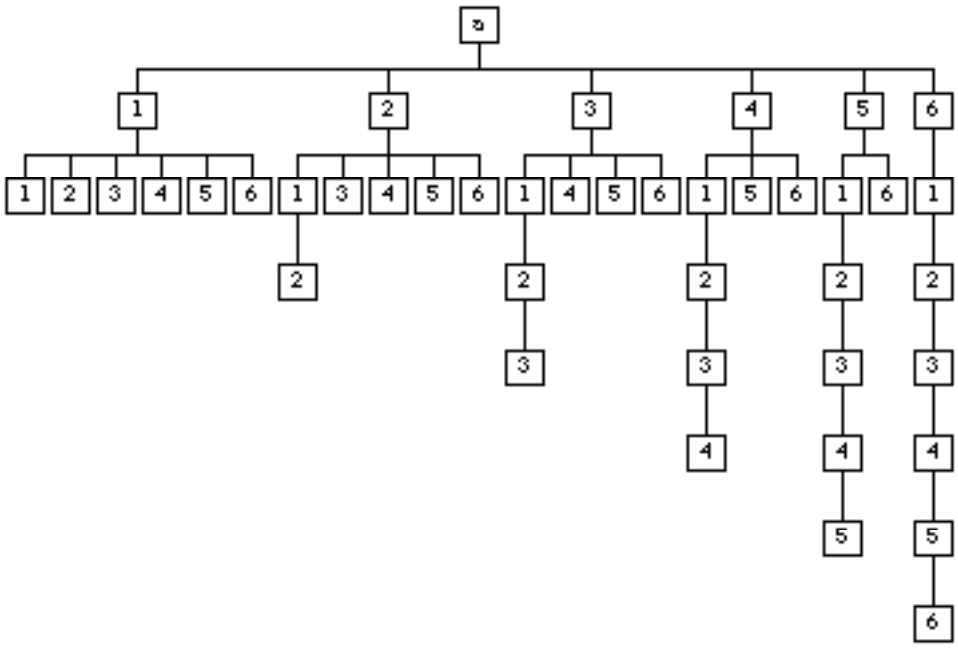

Figure 2.14: A "Debugger-Hostile" suspect tree

→

As a corollary, notice that for uniform suspect trees, with all nodes having the same number of children, GD&Q(1) would bring no advantage over AD&Q.

Regarding computational complexity, both AD&Q and GD&Q(1) require two linear suspect tree visits: one to obtain global information about the tree, and the second to pick the best node. The first visit can be amalgamated with the oracle refinement process. The whole process will therefore be O(n) (assuming direct accessibility of the suspect tree and oracle statements), except for the use of oracle rules for sophisticated refinement (such as subsumption), which will add to it.

We couldn't estimate accurately the advantage of GD&Q(N) (for N>1) over GD&Q(1). We conjecture it to be less appealing, considering the associated computational cost.

2.5. Domain heuristics

In addition to search-related heuristics, we may use information about the probability of an user answer. If, for a goal behavior facet, we can assign different probabilities for the user stating it correct or incorrect, we can adapt GD&Q(N) accordingly, to minimize the *expected* diagnosis cost. To the result we call *Probabilistic Divide and Query(N)*, or *PD&Q(N)*:

3''''. Consider the set S_N of debugger node descendents bist(F,OT) of the current node, that are N plies (2N levels) below in the tree, or higher up if they're leaves. Select a child of the current node, a bug node Q, minimizing the expected[1] h_f(F,OT) for any descendent of Q in S_N.

The probabilistic domain information can be gleaned from experienced human debuggers. In [68] several heuristics are proposed.

PD&Q(N) reduces to GD&Q(N) if the user answer probabilities are always equal (50% both for a "correct" or "incorrect" statement). Notice also that although the average performance of PD&Q(N) will in general be better than GD&Q(N)'s, the worst case performance, when the user states the unexpected, will degrade: more queries will be needed to reach a diagnosis.

2.6. Clever execution interpreters

So far we've been building a declarative debugging framework for generic SLDNF. But how dependent are we from each particular control strategy ? No matter what the execution strategy for each particular computation, as long as it is sound and complete regarding SLDNF, the present framework can be applied. On the other hand, some strategies are more interesting than others.

"**Clever execution principle**": given a bug manifestation T for a computation strategy A using a particular program, with suspect tree $ST_A(T)$, the diagnosis cost will be lower if we define $ST_B(T)$ from a smaller but equivalent computation B which avoids redundant computations. The roots of redundant subcomputations can be safely ignored, because there will always be a corresponding (non-redundant) subcomputation elsewhere which can be examined for debugging purposes.

We now examine two possibilities for taking advantage of this principle: "sidetracking" and "intelligent backtracking".

---

[1] By taking a weighted average of the estimative function in all nodes N plies below (or higher if leaves), with the combined probabilities of the query/answer sequences until them.

2.6.1. Sidetracking

An interpreter performing *sidetracking* orders goals according to their "determinism", i.e. their tendency to originate choicepoints with a small/large number of choices. The idea is to execute first goals that leave less choices open, such as for example a goal matching a single or no clause, so that failures tend to originate early in the execution, and less backtracking is needed to find a solution or to conclude that there isn't any. This idea was explored in [74]. It has been revived lately in connection with the Aurora language work, to better parallelize the execution of conjoined goals [88].

We can take advantage of such a mechanism by simply constructing a suspect tree from the computation trace under sidetracking, as long as the language we're using (say, normal programs) is supported by the sidetracking executor[1]. The parts of the suspect tree depending on failures will tend to be smaller, with less nodes and smaller tree branching factor, and less queries will be needed for a diagnosis. The parts corresponding to successful derivations won't bring any advantage, because the sidetracking mechanism preserves them, although possibly building them in a different order.

2.6.2. Intelligent backtracking

Backtracking being one of the first search strategies invented, it was one of the first to suffer improvements. "Non-chronological backtracking" was the term used in [85] for the strategy followed by an electric circuit constraint solver, using dependencies among variables to guide search, instead of performing blind or "naive" backtracking. Later this idea was generalized for logic programming, for example [75], [10]. And lately the first compiler-based implementation was done [18], promising the availability of efficient intelligent backtracking in future Prolog implementations.

---

[1] For example, the presence of Prolog cuts imposes an ordering for subgoal evaluation, inhibiting sidetracking in general.

The basic idea of intelligent backtracking, as applied to Prolog, is to follow the normal execution order of Prolog, but to change the way backtracking is done. Essentially, rather than blindly failing back to the previous choicepoint, if it is known that the present failure could not possibly be caused by the variable bindings done at that last choicepoint, then execution jumps back to some older choicepoint. In many cases this avoids the plaguing thrashing behavior of naive backtracking.

As with sidetracking, the advantage for debugging consists in having smaller and narrower (i.e. with smaller branching factor) suspect trees for failures, whenever intelligent backtracking avoids useless search[1]. Suspect trees for solutions are the same, except for parts affected by negation. The first application of intelligent backtracking to debugging, and the only one implemented, is [64].

Whereas an intelligent backtracking executor is concerned only in minimizing search using a correct program, its use for debugging requires additional care. In particular, whenever a goal match with a clause binds or simply tests a variable involved in a later unification failure, it must always be considered as suspect, even if there are no clauses left (i.e., if it is no longer a choicepoint). This requires information from the executor, and forbids some of its optimizations.

2.7. Relaxing the basic setting assumptions

We'll now review the assumptions we spelled out at the beginning, and examine how they affect the framework developed so far.

2.7.1. Inconsistent oracle

We assumed the oracle to be consistent. But in a practical setting, the oracle being based on a human, we must consider the case where the oracle “contradicts itself”.

---

[1] Two somehow similar situations are already “built-in” the debugging framework: since negated goals can't produce variable bindings affecting their brother goals, due to the negation as failure mechanism, when they succeed they're ignored in what concerns the missing solution problem, (cf. definition of incomplete predicate instance); and also in the oracle rule for solutions to goals that they subsume (cf. “Customized Oracle Theory” section).

Since the oracle theory contains statements about goal behavior facets, rather than about literals, and since the user is never asked to classify the same goal behavior facet twice, it is impossible to have directly contradictory statements. The other type of oracle inconsistency arises when the diagnosis obtained is unsatisfactory to the user:

```
o_s(incorrect,F)^ #RSS(F,OT)=1 ^ (F matches an instance D of
program component C) ^ "C is correct"
```

OT is the oracle theory including all user statements until the diagnosis D is obtained, and "C is correct" is a different type of oracle statement, akeen to correct_component(C) but not so firm. This type of oracle statement is obviously hard to obtain from the user (except in the form of the referred program specifications, or additional oracle knowledge, in which case such components will never be given as diagnosis): were it easy this thesis would be useless[1] ! The only situation where we consider obtaining it is for the user to complain about the debugger.

In order to retract the diagnosis, the user must be given the chance to retract one or more of his statements implying the diagnosis. In other words, he must refute the diagnosis, according to the bug instance definition which the debugger uses.

Let U be the relation u_s(Status,Facet) in OT, and OT'=OT\U. The user must change at least a statement in the relation U', a minimal subset of U such that o_s(incorrect,F)^ #RSS(F,U'≈ OT')=1. Typically U' will comprise the statements about literals appearing in the buggy instance D.

A practical declarative debugger must therefore provide "diagnosis explanation" and "oracle statement undo" facilities, allowing the user to examine U' and change it, respectively.

2.7.2. Infinite computations

Infinite computations are pervasive in many computer programs, irrespective of the programming language. Logic programming is no exception.

---

[1] Because if the user could classify program source components as correct or incorrect, he wouldn't need to use a debugger.

We conjecture that there are infinite computations for which it is not possible to have a diagnosis along the lines of the present chapter. We defined diagnosis in terms of declarative semantics, whereas infinite computations stem from the operational semantics.

For example, a repeated recursion pattern may be automatically detected in a buggy nonterminating computation, but it may be impossible to refine the set of goal suspects in that pattern, at least with declarative information alone. Other authors [83] have used non-declarative extensions to the oracle: this assumes the availability of a *well-founded ordering of goal calls* - an operational concept.

There are of course cases where a wrong or missing solution with a finite subcomputation can be detected[1]. In such cases a bug can be found with any of the previous algorithms. A practical debugger must therefore provide inspection facilities for interrupted computations[2].

2.7.3. Floundering computations

A floundering computation (a computation selecting a non-ground negative literal) is usually regarded as an error[3]. Which raises the question of its declarative debugging.

Floundering occurs when *all* goals in an SLDNF-tree node are nonground negative literals. If the node has a single negative literal, floundering can be regarded as a particular case of inadmissible goal call (cf. next chapter), and hence a bug instance can be found. This can be useful for systems without goal delaying, like most existing Prolog systems. We have no solution to the general problem however.

2.8. Relationship to declarative semantics

The framework developed so far, in particular the part related to the oracle theory, maps directly into standard declarative semantics concepts:

---

[1] Or an inadmissible goal call (cf. next chapter).

[2] "Infinite" computations are typically detected by the user, who presses a "ctrl-c" key or something similar, after having waited in vain for its normal completion.

[3] It could also be regarded as a feature rather than a nuisance, for example by considering the (floundering) negated goals as informative constraints on its variables.

Declarative Semantics	**Our Framework**
Intended model	Oracle theory
False solution G	o_s(incorrect,solution(G)) is true
Missing solution G	o_s(incorrect,solution_set$_{\varnothing}$(G)) is true
False clause	Wrong clause instance
Uncovered goal G	Incomplete predicate instance matching G

Table 2.4: Relationship to declarative semantics

The reasons why we decided to develop our own framework, rather than just using standard semantic concepts, are essentially the following:

• To abstract from the type of bug manifestation, concentrating on relevant common aspects, and thus develop a uniform diagnosis framework.

• To tackle impure logic programming dialects and the new bug manifestations they originate, such as in Prolog; this will surface in subsequent chapters.

3. Other framework issues

In this chapter we discuss some improvements and adaptations to the previous framework, but still without introducing Prolog's main impurities. The improvements are: to debug type violations, to allow refined oracle statements, and to deal with meta-interpreters and transformed programs.

3.1. Inadmissible (type-violating) goals

In this section we'll introduce a new type of bug manifestation, for logic programs using partial relations - inadmissible goals. This was first defined in [64], and later revisited in [68], where we presented an approach similar to the one below.

3.1.1. Type violations

Wrong clauses and incomplete predicate definitions are unsatisfactory as diagnosis in many situations. This happens because logic programmers usually leave many details implicit in predicate definitions. Namely, they tend to express *partial* relations, leaving undefined a substantial part of the relation's domain because they expect it to be unnecessary for the use of the program.

Example Consider the following program:

```
top(L) :- append([a,b],not_a_list,L).

append([],L,L).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

Assume that for goal top(L), the solution top([a,b|not_a_list]) is considered wrong. The traditional declarative debugging approach gives us as diagnosis one of the clauses of `append`, because `append` does not check that its second argument is a list. It will accept any term instead! →

Rather than adopting this traditional purist philosophy, expecting programmers to provide exaustive and cumbersome case analysis, it seems more reasonable to accept the practical status quo, and to provide mechanisms to find the real bug in the previous example: the `top` clause has an *inadmissible subgoal*, because it makes an *inadmissible goal call* to `append` which violates the implicit data type it supports.

3.1.2. Inadmissible goal calls

We now define the changes to the oracle framework. The basic idea is that now goals can be inadmissible or admissible; and if they're inadmissible, the correctness status of their solution set is undefined.

An *inadmissible goal call* is a goal call which is considered by the oracle as a bug manifestation, irrespective of calling context as well as of its solutions. In the previous example, `append([a,b],not_a_list,L)` is an inadmissible goal call. An admissible goal call is one which is stated not inadmissible.

This brings attention to goal calls, turning them into an additional type of goal behavior facet. Here's our updated table for oracle statements:

Type	**Status**	**Meaning**
call	correct	goal is admissible
call	incorrect	goal is inadmissible
solution	correct	solution is correct for an admissible goal call
solution	incorrect	solution is incorrect for an admissible goal call
solution set	correct	all solutions for the admissible goal are correct, and none is missing
solution set	incorrect	the solutions for the admissible goal are correct, but some are missing

Table 3.1: Oracle framework for inadmissible calls

Since inadmissible goals have undefined correctness status for their other facets, we should add the following rule to the customized oracle theory, expressing the absence of statements about them:

o_s(incorrect,call(G)) ∅

¬ o_s(_,solution_set(G)) □

(solution(call(G),N,_,_) ∅ ¬o_s(_,N)).

And also a rule to use subsumption between goal calls:

subsumed_facet(call(G1),call(G2)) □ “G1 subsumes G2”.

Consequently (cf. Core Oracle Theory above), a goal call subsumed by an admissible call is automatically found admissible; and a goal call that subsumes an inadmissible call is automatically found inadmissible.

3.1.3. Another bug type: inadmissible subgoal instances

We now define an additional type of bug instance, inadmissible subgoal instance, motivated by inadmissible goal bug manifestations.

Let A be a goal matching clause instance (H <- B1…,Bi,…Bn). Bi is an *inadmissible subgoal instance* if:

• o_s(correct, call(A)) and o_s(incorrect,call(Bi)) are true.

• Consider the SLDNF-tree for the computation producing the goal call Bi, and the path from the A node down to Bi. For each literal Bj (j≠i) in the body which is selected in a node in the path, o_s(correct,solution(Bj)) is true.

With Prolog's left-to-right execution order, the previous Bjs will be simply those to the left of Bi.

We now move to suspect trees for inadmissible goals.

3.1.4. Debugging inadmissible goals

We'll define a suspect tree identical in form to those for wrong solutions and incomplete solution sets. Only the node names, being unique among all goal behavior facets, will be different, as well as the children of the new node type.

The *suspect tree* for an inadmissible goal call I, ST(I), is based on the SLDNF-tree of the computation producing it, and is defined like the suspect trees for other facet types, with the following differences:

• The root is the node I.

• Each call facet node G is labelled with its father clause instance C[1], and has the following children:

  •• ST(A), where A is the goal call matching C (G's goal father).

  •• For each goal solution Bi in C's body, whose goal call is selected in one of the nodes in the SLDNF path from A to G, a child ST(Bi).

• Subtrees for solutions and solution sets are defined as before.

---

[1] I.e., such that goal(G,_,C,_) is true in the debugger theory.

Suspect trees for goal calls resemble those for solutions, except that the tree looks inverted and may correspond to a partial (interrupted) computation.

Example Consider the following program to count the number of integers in a list:

```
element_count(X,I):- integer(X), !. % bug: should be 1 instead of I
element_count(X,0).
count_int([],0).
count_int([Y|Z],N):-  element_count(Y,NY),  count_int(Z,NZ),  N  is
     NY+NZ.
```

On execution of top goal count_int([5,b],N), the goal "N is I+0" will be called, violating the type requirements of the arithmetic built-in predicate `is`. Here's ST(call(N is I+0)), layed down horizontally to save space here, and without the labels of program component instances. The system built-ins `integer` and `is` are assumed correct, and therefore do not originate suspects.

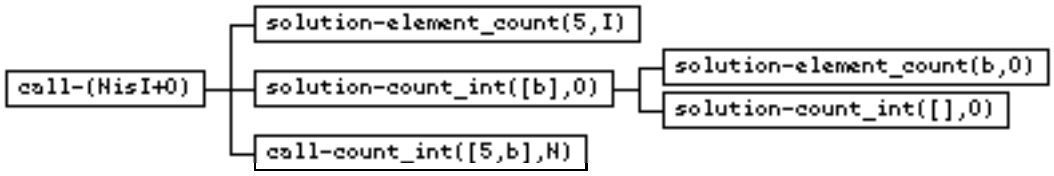

Figure 3.1: Suspect tree for an inadmissible call

The computation leading to "N is I+0" is suspect, including the top goal call (the user could have given a type-violating top goal!). →

The previous debugging framework can be used directly (suspect set definitions, debugging algorithms, etc.), by referring to the new definition for suspect trees and using the new oracle. With just a slight difference, patent in the following proposition.

Proposition 9 Consider an inadmissible goal G, under an admissible goal A. Then either there is a bug instance in SS(call(G)), or there's another inadmissible goal G' under A, labelling a node in ST(G).

Proof Almost as for proposition 1 (cf. section "Existence of a bug instance in a suspect set" in chapter 2). If the oracle stated that the goal call G' for one of the *solution* or *solution set* nodes in ST(call(G)) was inadmissible, than that goal call could not be an ancestor of G, and therefore its father predicate would not be in SS(call(G)). →

We're still sure of finding a bug instance, however, because SS(call(G')) is smaller than SS(call(G)), and the proposition applies.

This approach assumes that inadmissible goals are much less frequent than the other bug manifestations, or that typically they're caused by them. The bug instance search tree, and the algorithms based on it, depend on the suspect tree definition. Since this relegates inadmissible goals to a second plan, by including from the start only the *ancestors* of the first inadmissible goal, the debugging algorithms will also plan queries accordingly.

Example Here's the sequence of queries/oracle statements constructed by AD&Q for the previous `count_int` example, until diagnosis:

Goal behavior facet term	**Type**	**Correct ?**	**Comments**
N is I+0	call	no	detected automatically by the debugger, because the expression is not ground
count_int([b],0)	solution	yes	the node in the suspect tree which better “splits” it
element_count(5,A)	solution	no	because A is a free variable; diagnosis is found: the wrong clause instance matching this solution.

Table 3.2: Debugging an inadmissible call

→

3.2. Intensional oracle statements

Assume the user is able to state that all goal behavior facets satisfying some property P are incorrect. Such *intensional* user statements can be applied to different goal behaviors, whether or not there is a subsumption relation among them. Then the debugger may be able to refine its current suspect set, because it knows that the suspect set for *any facet satisfying P* contains a bug - in particular, one whose suspect set is smaller.

For instance, let the user statement be "all goal solutions containing term my_type(aa) are incorrect", say because the argument of the my_type functor must be an integer. Then the debugger may be able to refine its current suspect set more than by just using the statement "p(..,my_type(aa),...) is incorrect".

Example As a concrete example, take the following program, defining a predicate that transforms a Prolog term into a tree t(Number,Functor,Children):

```
number_term(T,N1,N3,t(n1,F,C)) :- % bug: should be N1 instead of n1
    T=..[F|Args], N2 is N1+1,
    number_term_list(Args,N2,N3,C).
number_term_list([],N,N,[]).
number_term_list([A1|An],N1,N3,[C1|Cn]):-
    number_term(A1,N1,N2,C1),
    number_term_list(An,N2,N3,Cn).
```

Here's the suspect tree for wrong solution `number_term( a(b,c), 1, 4, t(n1,a,[t(n1,b,[]),t(n1,c,[])]) )`:

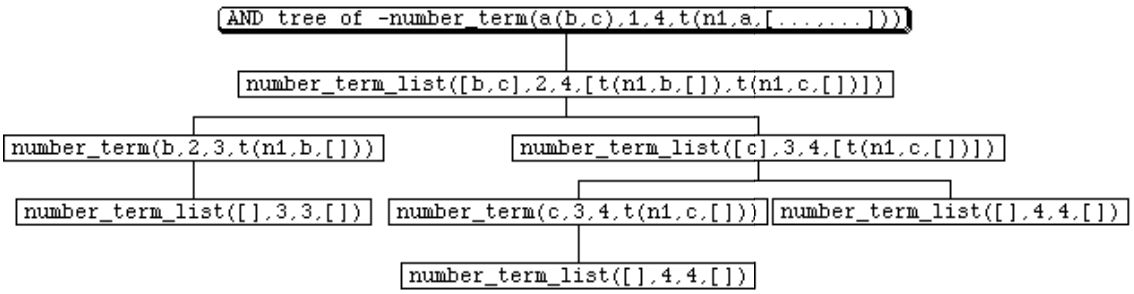

Figure 3.2: A suspect tree

With ordinary oracle statements, 3 queries are needed by AD&Q to find the bug instance:

Goal behavior facet term	Type	Correct ?	Comments
number_term(a(b,c), 1, 4, t(n1,a,[t(n1,b,[]),t(n1,c,[])]))	solution	no	top goal solution
number_term_list([c],3,4,[t(n1,c,[])])	solution	no	
number_term(c,3,4,t(n1,c,[]))	solution	no	
number_term_list([],4,4,[])	solution	yes	

Table 3.3: A query/answer sequence

If instead the user started by giving an intensional statement about the incorrectness of the top goal solution, for example "all solutions containing the term t(n1,_,_) are wrong", AD&Q would need only 1 additional query:

Goal behavior facet term	**Type**	**Correct ?**	**Comments**
number_term(a(b,c), 1, 4, t(n1,a,[t(n1,b,[]),t(n1,c,[])]))	solution	no, neither is any solution containing subterm t(n1,_,_)	top goal solution
number_term_list([],4,4,[])	solution	yes	Diagnosis found, because the debugger already knows that number_term(c,3,4,t(n1,c,[])) is incorrect, since it contains the term t(n1,c,[])

Table 3.4: Debugging using intensional statements

→

In theoretical terms, an intensional statement is just an additional clause to the o_s relation, or an "oracle assertion" (along the lines of [25]). Although originally motivated by [64]'s "wrong term" statements, intensional statements like that of the previous example have little relation with the former, which were used as an heuristic to guide search within a fixed suspect set, rather than to immediately refine suspect sets.

The HyperTracer supports a restricted form intensional statement at a time, in the form of a filter, which specifies a term that must be present in all suspects[1].

[1] There's no significant technical difficulty in implementing intensional statements as in the example above, we simply had no time for doing it.

3.3. Meta-interpreted programs

The use of meta-interpreters is one of the most appreciated features of logic programming. An interpreter provides the ability to obtain additional information, exert additional control on a "conventional" logic program, usually referred to as the *object program*, or for writing the latter in some different language, for which the interpreter provides the semantics.

Bugs in interpreters can be found using the framework developed so far. But given a *correct* interpreter, i.e. whose predicate definitions are not buggy, how can we find bugs in the object program ? Simply by taking into account that interpreters are logic programs, and assuming object programs to be codified with the standard auxiliary "`clause`" relation accessible by the interpreter.

The approach is as follows: given a bug manifestation of the object program, via a result computed by the interpreter, apply any declarative debugging method as usually; the user is assumed able to answer any query about correctness of any interpreter *goal behavior facets whose suspect set includes (via* `clause`*) some object program component*. Other interpreter goals are necessarily known correct, given that the interpreter is correct. But the user is *not* assumed to be able to answer about the correctness of results of `clause` goal calls[1].

Eventually a program component instance of the interpreter will be found whose head is incorrect and whose body will be wholly correct, apart from one or more results of calls to `clause` (let's call it a *meta bug instance*). This follows easily from proposition 1. Given that the interpreter program component is correct, some of the `clause` results must be incorrect, meaning that one of the respective object program components is buggy. But it won't be possible in general to choose a single one as the culprit.

Example Following is a trivial Delta-Prolog [20] program:

```
b(2). % bug: should be b(1)

foo(X) :- ( (X?ev // Y!ev), b(X) ).
```

[1] If that was the case, the user would certainly not need a debugger to find the bug! He would simply take a look at the object program source to find the incorrect component…

And now one of the clauses of a sequencial interpreter for Delta-Prolog[1], the only one calling the `clause` meta-predicate:

…

```
i(G,NG,E,Tail,New_Tail) :-
      clause(G,B), i(B,NG,E,Tail,New_Tail).
```

…

The solution for call foo(X) is foo(2). Assume that foo(2) and b(2) are incorrect. After some oracle queries, the debugger will have isolated the following clause instance as a likely bug instance (the head being stated as incorrect by the oracle, and the second subgoal as correct):

```
i(b(2),true,E,Tail,Tail) :-
      clause(b(2),true), i(true,true,E,Tail,Tail).
```

`clause(b(2),true)` will therefore be concluded incorrect, and the first clause of the Delta-Prolog program to be the bug. →

As the example shows, a crucial issue is to avoid showing irrelevant detail to the user. Whenever possible, the declarative semantics of the object program language should be used to build simpler queries. Instead of querying the oracle “Is i(b(2),true,E,Tail,Tail) a correct solution ?” it would be better to query “Is b(2), with empty event trace, a correct solution ?”.

It should be possible to provide such “pretty printing” of queries for most object program languages, using information provided in the interpreter's arguments. But in order for our approach to apply, the “pretty-printing relation” between debugger queries DQ_i and their prettier and higher-level counterparts PQ_i, must satisfy the following constraints:

DQ_i incorrect × PQ_i incorrect

DQ_i correct × PQ_i correct

---

[1] The interpreter is listed in appendix B.

Although we're concerned with showing a query in a higher level form closer to the object program semantics, but not necessarily with hiding or abstracting details, our pretty-printing relation still reminds of [47]'s abstraction function, in the sense that both map a query into another more suitable to the user. Their definition must verify a less strict condition, which in our setting would be written as:

DQ_i incorrect ♦ PQ_i incorrect

This weaker constraint forces them to make "concrete queries" (our DQ_is) after having localized an "abstract diagnosis" (our "meta bug instance"): it may be the case that some computation results declared correct via their "abstract" (our "pretty") counterparts are incorrect after all. In principle we could also adopt this strategy, and make "ugly" queries at the end. We didn't because:

1) In many cases it is easy to obtain a pretty printing relation obeying our constraints.

2) We envisage having users which ignore even the existance of the meta-interpreter, to say nothing of its semantics. For example, a linguist developing a natural language grammar executable by a meta-interpreter.

3.4. Pre-processed programs

Another useful technique, more efficient[1] although less flexible than the use of meta-interpreters, is preprocessing, or program source transformation. The transformation can be either entirely automatic, as in partial evaluation [50], or specified by a preprocessor implemented for a particular case, for example the definite clause grammar translator present in most Edinburgh Prologs. Conceptually, preprocessing is just a way of embedding the semantics of some meta-interpreter into an object program - in the present context better referred to as the *original* program.

Like an interpreter, a preprocessor can be debugged using the methods for ordinary programs. The oracle will just need to have in mind the intended semantics for it. But how to find a bug in an object program, while querying the oracle about goal behavior facets of the transformed program ?

---

[1]Usually the preprocessed program can profit more from compilation than would be the case for an interpreter+object program.

First, to each transformed program component C (a clause or predicate definition) we associate a particular subset of the components of the original program, the *origins set*: those and only those components of the program necessary for the translation of C. This can be easily computed automatically by the preprocessor loading the program.

Then we simply use any of the previous debugging algorithms, to find a buggy component of the transformed program. This assumes the oracle to be able to answer all queries about transformed program goals. When a bug instance is found, instead of presenting it to the user we present the origins set. If this happens to be a singleton, a single buggy program component can be pinpointed in the original program, but in general the diagnosis will consist in more than one.

As for interpreters, if a pretty-printing relation can be defined, obeying the same constraints, the queries become simpler to answer by the oracle.

Example The HyperTracer supports Definite Clause Grammars [62], by using a simple pretty-printing function and by keeping the origins set for each transformed clause - a single DCG rule. For pretty printing of a non-terminal, the difference list is represented separated from the goal arguments, as an ordinary closed list of the terminals consumed under the non-terminal (the dots inside the functor sentence mean that larger subterms are inside, which can be seen with the graphical term browser; it has nothing to do with the DCG pretty-printing):

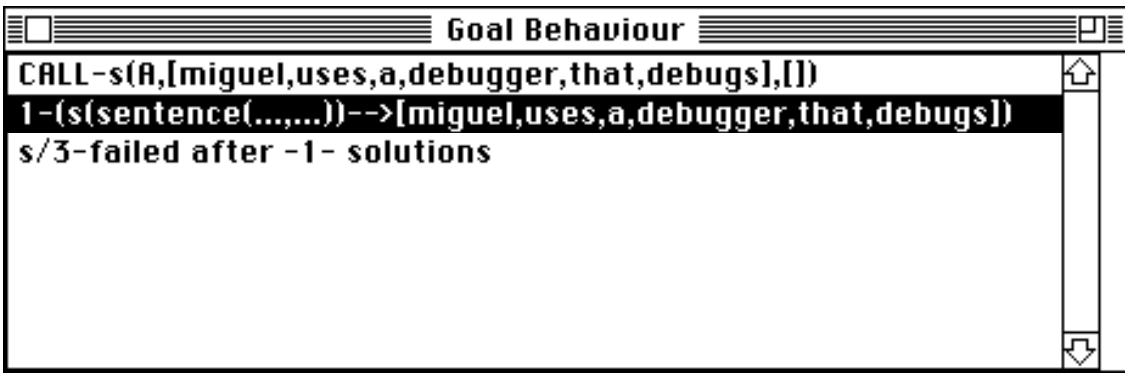

Figure 3.3: Pretty-printing a DCG goal solution

For diagnosis, the origins set is used, simply by associating with the transformed clauses a reference to the textual DCG rule; for example:

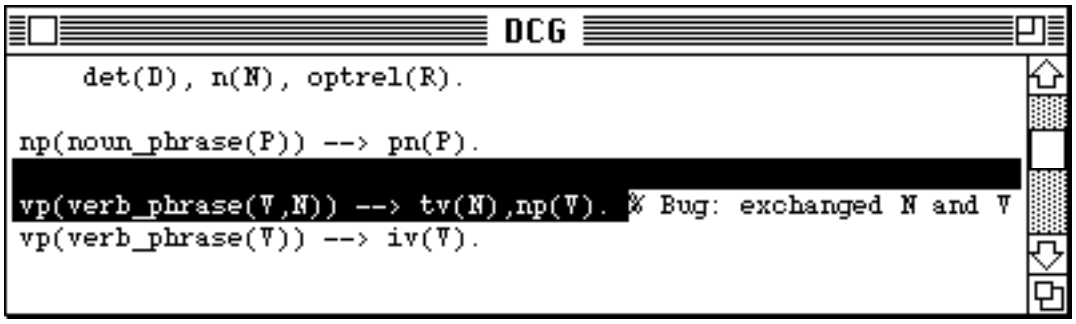

Figure 3.4: Showing a bug in a DCG

→

4. Extensions for impure logic programming

In this chapter we'll leave the safety of normal logic programs and venture into the use of impure Prolog features. We'll extend the current framework in order to debug programs using such features, but still relying on declarative information alone.

The main results are:

• Extensions to the suspect sets, and new bug instance definitions, to cater for the effects of *cuts*.

• The definition of a new bug manifestation, "wrong output segment", and additions to the suspect trees, to treat *output side-effects*.

• Further extensions to the suspect sets, to treat partially bugs due to the use of *internal database side-effects*.

4.1. Cuts

Our approach consists in extending suspect sets to cater for the effect of cuts, while keeping nearly the same declarative oracle framework. It just happens that, in general, as we add impure features to the logic programming language, suspect sets become larger.

This approach was first presented in [68]. There we used modified SLDNF trees to define the effect of cuts ("behavior justifications"), and also to define suspect sets. Here we'll use a more operationally-oriented description, which we hope is more easy to follow.

4.1.1.Changes to the oracle

How does a logic program with cuts behave erroneously ? The same way as usual - by producing erroneous computation results. However, cuts introduce some complications.

First, there's the trivial case of "green cuts" [35], those cuts whose prunning does not prevent legitimate derivations to be derived from the program (also called "safe cuts" in [49]). Such cuts are merely present for control, and have no declarative meaning. Therefore they could be ignored regarding debugging, (cf. [83], p. 76). The issue remains however [ib.] of distinguishing green cuts from "red cuts" (or "unsafe cuts", [49]), those prunning search paths leading to successful derivations, but which would be logically legitimate only if we gave no logical meaning to cuts.

Red cuts cause the answer set of a program to depend on the instantiation of goal calls. Therefore, in order to describe the intended semantics of the program we need <goal call, goal solution> pairs, rather than just goal solutions as for pure normal programs.

Example (Adapted from [51])

p(X,Y) :- !, X=2,Y=2. % should be X=1, Y=1

p(2,Y) :- !, Y=2.

Calling p(X,Y) results in solution p(2,2), which is a correct solution according to the traditional semantics. But the "expected" solution would be p(1,1) ! →

Rather than ignoring such bug manifestations, we prefer to redefine our oracle framework to cater for them. For the above example the correct solutions would be specified as {<p(X,Y),p(1,1)>, <p(2,Y),p(2,2)> }. Variables in this specification are no longer logical variables, just instantiation patterns.

We'll now enumerate the necessary **changes to the oracle framework**. First, we revisit the table with the goal behavior facet types, and the possible bug manifestation cases.

Type	**Status**	**Meaning**
call	correct	goal pattern is admissible
call	incorrect	goal pattern is inadmissible
solution	correct	solution is correct (wrt a particular goal call)
solution	incorrect	solution is incorrect (wrt a particular goal call)
solution set	correct	all solutions for the goal are correct (wrt it) and none is missing
solution set	incorrect	the solutions for the goal are correct (wrt it), but some are missing

Table 4.1: Oracle framework for cuts

These are the changes:

• The admissibility of goal calls specifically takes into account their term pattern, i.e., the call bindings[1]; there's no subsumption relation among calls (cf. below).

• The qualification of a solution's correctness with respect to the goal call.

• Allow statements about solution sets (i.e., their completeness) only for goals that *failed completely*. By this we mean that their execution was not prunned by a cut to one of their ancestors, whether the goal produced solutions or not[1].

[1] Our admissible goals now ressemble Bonnier's “admitted” goals [6], a concept he defined to build a semantics for Prolog built-in (i.e., nonlogical) procedures.

The remaining core oracle theory still applies. The customized oracle theory suffers an important change. The rules for subsumed_facet now become:

subsumed_facet(solution(S1),solution(S2))
"S1 identical to S2" "call(S1) identical to call(S2)"

subsumed_facet(call(G1),call(G2)) "G1 is identical to G2".

Since the intended semantics is now expressed with input/output relations between term instantiation patterns, logical subsumption can no longer be used.

This fact also invalidates a customized oracle rule: a solution identical to its goal call does not necessarily "subsume" other more specific solutions, with respect to the intended semantics of the program.

4.1.2. Bug instances with cuts

The debugging algorithms presented so far can be employed straightforwardly, using instead the modified oracle. But whenever a bug instance is found in a predicate definition containing cuts, it may be found "inadequate" (cf. below).

4.1.2.1. Wrong clause instances

Here's an **example** of an inadequate diagnosis obtained with the previous debugging algorithms[2].

```
(1)   h(X,a) :- c(X), !.
(2)   h(X,b).
(3)   c(3). % bug: missing clause c(1).
```

---

[1]It would be pointless to assign guilt to a symptom which could originate in the goal's context. In other words, one would be asking *non-declarative* information from the oracle, by asking the latter to qualify the correctness status to the precedents of a goal call - e.g., "there are no missing solutions to goal G1 because the clause calling it has a cut to make G1 deterministic". Declarativeness implies that the oracle statements are valid no matter what the goal's context.

[2] Or with any algorithm from other authors, except for [40], developed independently, and our own, as shown at the ESPRIT project 973 review meeting, Paris, October 1987.

Goal call h(1,X) will give solution h(1,b). Assume this to be a wrong solution. Then a declarative debugger, as described in the previous chapters, would return h(1,b) as a wrong clause instance (or clause (2) as a wrong clause, for a declarative source debugger). Whereas it should return <c(1), c predicate> as an incomplete predicate instance!

It is thus necessary to *redefine* bug instances, and subsequently suspect trees, sets, etc., in order to find adequate diagnoses. These new definitions assume Prolog's left to right execution strategy, and thus we use expressions like “to the left of a cut”.

A *wrong clause instance* expresses the notion of buggy clause. Let H be a goal solution for a goal G, matching clause instance H<-B1…Bn. This is a wrong clause instance if:

• o_s(incorrect, solution(H)) is true, and for each literal Bi in the body , o_s(correct,solution(Bi)) is true. *And*:

• Let PB be the predicate body instance for G. For all goal calls Gi in PB, which are textually to the left of a cut, have failed completely, and occur in the clause or in previous ones in the predicate, o_s(correct,solution_set(Gi)) is true.

Such goal calls, if successful, could prevent the buggy clause being used. In the previous **example**, clause (2) would not be considered buggy without additional information - unless the solution set for goal call c(1) was stated correct.

In operational terms, it's as if clauses in a program with cuts were preprocessed, to insert the negations of the subgoals preceding cuts in the previous clauses of the same predicate, together with the relevant inequalities involving head arguments. For the previous example, where X is a head variable without multiple occurrences, the we'd simply end up with:

```
(1')  h(X,a) :- c(X).
(2')  h(X,b) :- ~c(X).
(3')  c(3).
```

4.1.2.2. Inadmissible subgoals

Similarly, we must change the definition for the inadmissible subgoal bug instance, now assuming Prolog's left-to-right execution order.

Let A be a goal matching clause instance (H <- B1…,Bi,…Bn). Bi is an *inadmissible subgoal* if:

• o_s(correct, call(A)) and o_s(incorrect,call(Bi)) are true.

• Consider the SLDNF-tree for the computation producing the goal call Bi, and the path from the A node down to Bi. For each literal Bj in the body to the left of Bi, o_s(correct,solution(Bj)) is true.

And now the additional case to consider when cuts are present:

• Let PB be the predicate body instance for A. For all literals Gi in PB, which are textually to the left of a cut in the predicate, have failed completely, and occur in the above clause or in previous ones in the predicate, o_s(correct,solution_set(Gi)) is true.

This corresponds to the second case in the definition of wrong clause instance above.

4.1.2.3. Incomplete predicate instances

Similarly, we must change the definition for "incomplete predicate instance". Not the least because solution sets have been defined for goals which fail completely, i.e. exaust their solutions, whereas with cuts this may not happen, whenever an ancestor cut does its prunning. But also because a cut may (erroneously) prevent alternatives to be found.

Example For goal call h(2,Y), the following program fails to produce the expected solution h(2,b):

```
(1)   h(X,a) :- c(X), !, b(X).
(2)   h(2,b).
(3)   c(2). % bug: wrong clause.
(4)   c(3).
```

The goal c(2) has its execution prunned by the cut in clause (1), and doesn't "fail" - it's "skipped over" on backtracking. And although previous declarative debuggers would return predicate h/2 as an incomplete predicate instance (assuming b(2) to be legitimately failed), the diagnosis should be instead "clause (3) is wrong". →

An *incomplete predicate instance* expresses the notion of buggy completion rule, or of a predicate whose clauses fail to produce an additional solution without any "guilt" being assigned to subgoals. It is *defined only for goal calls which failed completely*, i.e. had no search paths pruned by brother or uncle cuts. It is a predicate instance <G,P>, for which o_s(incorrect, solution_set(G)), and such that for all goal calls in its predicate body instance, named Gi:

• if Gi is a completely failed atomic goal, then o_s(correct,solution_set(Gi)) is true

• if ~Gi is a negative literal with empty solution set, then o_s(correct, solution_set∅(~Gi')) is true for all solutions Gi' to Gi.

• if Gi is an atomic goal that didn't completely fail, because it occurs textually to the left of a cut that was reached during execution, then o_s(correct,solution(Gi')) is true for each solution Gi' of Gi.

• if ~Gi is a successful negative literal, with solution ~Gi', occurring to the left of a cut that was reached during execution, then o_s(correct,solution(~Gi')) is true.

• (Other successful negative literals are irrelevant, as in the original definition).

Notice that successful negative literals are no longer irrelevant if they lead to a cut that prunned alternatives. In general, goal solutions leading to a cut that was reached must be stated correct for the enclosing predicate to be considered buggy.

In the previous **example**, predicate h/2 would not be considered buggy without additional information - unless the solution c(2) was stated correct.

4.1.3. Suspect trees for computations using cuts

We can now proceed adapting the suspect tree definitions, taking care to preserve their properties regarding the new bug instance definitions.

The *suspect tree* for a goal behavior facet F, ST(F), is based on the SLDNF-tree of the computation producing F, which is not necessarily at its root, and is recursively defined as follows:

• Each node has the name of a goal behavior facet, and is labelled with a program component instance

• The root is the node F.

• Each *solution* facet node S is labelled with the clause instance matching it and, for each literal Bi in the clause instance body, it has a child as follows:

  •• if Bi is an atom, ST(solution(Bi))

  •• if Bi is a negative literal ~G, ST(solution_set∅(G))

In addition to these, *the node may have additional children due to cuts*. Let PB be the predicate body instance for the goal producing solution F, and P the predicate. For all literals Gi in PB, which are textually to the left of a cut, have failed completely and occur in the clause or in previous ones in P, the node has a child as follows:

•• if Gi is an atom, ST(solution_set(Gi)).

•• if Gi is a negative literal with no solutions, ~G, a child ST(solution(G')) for each solution G' to G.

• Each *call* facet node G is labelled with its father clause instance C[1], and has the following children:

•• ST(A), where A is the goal call matching C (G's goal father).

•• For each goal solution Bi in C's body, whose goal call is selected in one of the nodes in the SLDNF path from A to G, a child ST(Bi).

In addition to these, *the node may have additional children due to cuts*. Let PB be the predicate body instance for A, and P the predicate. For all literals Gi in PB, which are textually to the left of a cut, have failed completely and occur in the clause or in previous ones in P, the node has a child as follows:

•• if Gi is an atom, ST(solution_set(Gi)).

•• if Gi is a negative literal with no solutions, ~G, a child ST(solution(G')) for each solution G' to G.

• Each *solution set* facet node is labeled with the predicate instance matching it, and, for each literal Bi in its predicate instance body, i.e. all goal calls immediately *under* it, it has one or more children as follows:

•• if Bi is a completely failed atom, a child ST(solution_set(Bi)).

•• if Bi is a negative literal ~G without solutions, a child ST(solution(G')) for each solution G' of G.

In addition to these, *the node may have additional children due to cuts*.

---

[1] I.e., such that goal(G,_,C,_) is true in the debugger theory.

•• if Bi is an atomic goal that didn't completely fail (because it occurs textually to the left of a cut that was reached during execution), a child ST(solution(Bi')) for each solution Bi' of Bi.

•• if Bi is a successful negative literal ~G occurring to the left of a cut that was reached during execution, a child ST(solution_set$_{\varnothing}$(G)).

It's easy to see that all propositions referring suspect trees still hold.

Example Following is a (buggy) program to compute prime numbers:

```
primes(Limit,Ps) :- integers(2,Limit,Is), sift(Is,Ps).

integers(Low,High,[Low|Rest]) :- Low =< High, !,
    M is Low+1, integers(M,High,Rest).
integers(_,_,[]).

sift([],[]).
sift([I|Is],[I|Ps]) :- remove(I,Is,New), sift(New,Ps).

remove(P,[],[]).
remove(P,[I|Is],Nis) :- is_multiple(I,P), !, remove(P,Is,Nis).
remove(P,[I|Is],[I|Nis]) :- remove(P,Is,Nis).

is_multiple(I,P) :- 0 is P mod I. % bug: exchanged I,P
```

For goal primes(4,P), it gives the solution primes(4,[2,3,4]), which is wrong because 4 is not prime.Its AND tree (also the suspect tree, if there were no cuts in the program) follows:

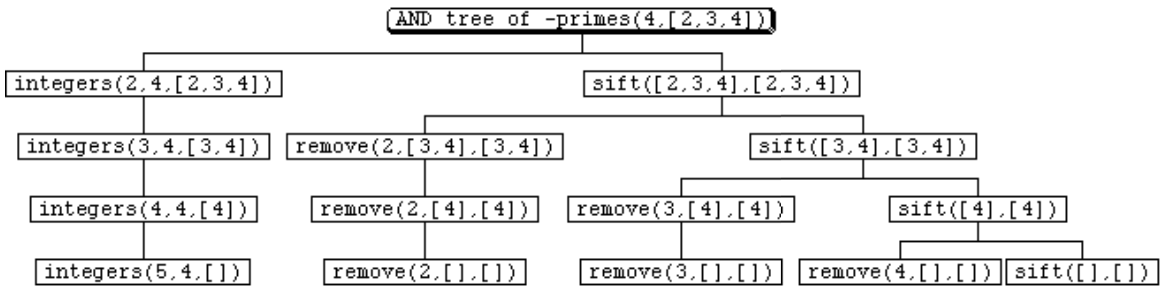

Figure 4.1: Suspect tree for a wrong solution

The effect of cuts consists in adding some suspects, corresponding to the failures of calls to `is_multiple`:

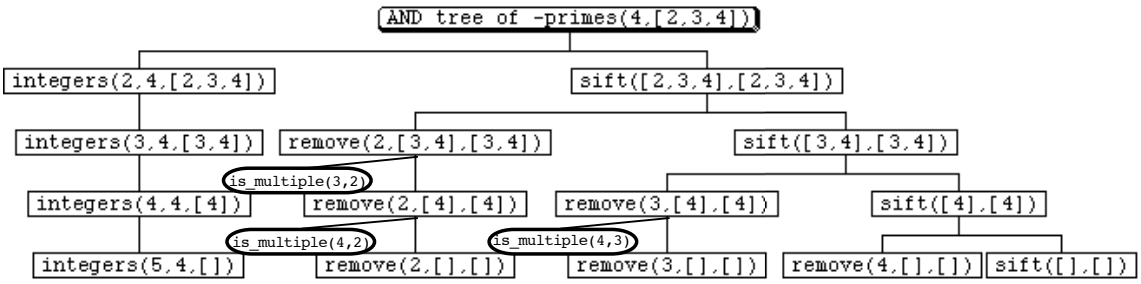

Figure 4.2: A suspect tree, considering cuts

→

4.1.4. The debugging framework for programs with cuts

Having changed the oracle, as well as the bug instance and suspect tree definitions, *no other changes are necessary*! Suspect sets, suspect set refinement, bug instance search trees, and the different algorithms are all immediately usable once they are applied using the changed concepts above.

Example Here's the debugging session using AD&Q in the HyperTracer debugger, for the suspect tree of the previous `primes` example. It starts with a complaint about the top goal solution, using one of the HyperTracer's "WHY" commands (cf. chapter 6):

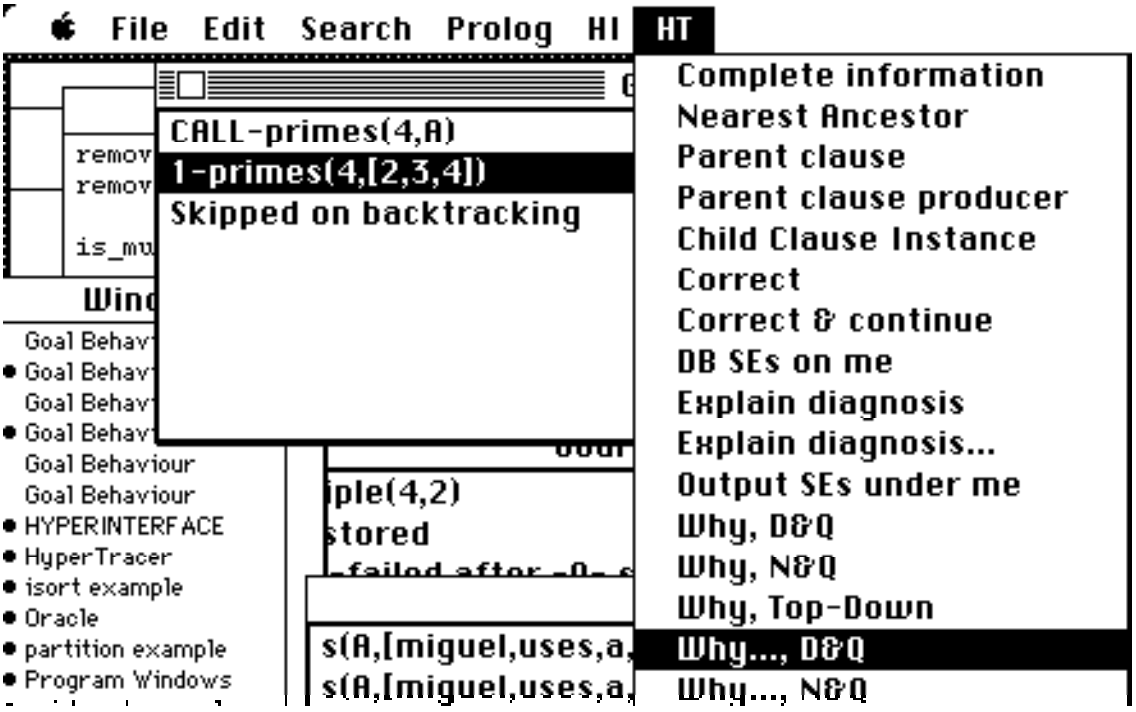

Figure 4.3: Debugging with cuts

The queries follow, each corresponding to a new goal behavior window:

Goal behavior facet term	**Type**	**Correct ?**	**Comments**
primes(4,[2,3,4])	solution	no	top goal solution returns a list containing 4, which isn't prime
sift([3,4],[3,4])	solution	yes	
remove(2,[3,4],[3,4])	solution	no	4 should be removed from the list
remove(2,[],[])	solution	yes	
remove(2,[4],[4])	solution	no	
is_multiple(4,2)	failure without solutions	no	And diagnosis is reached, "`is`" being a correct system predicate

Table 4.2: Debugging with cuts

Finally, the last query is made, by showing the following window:

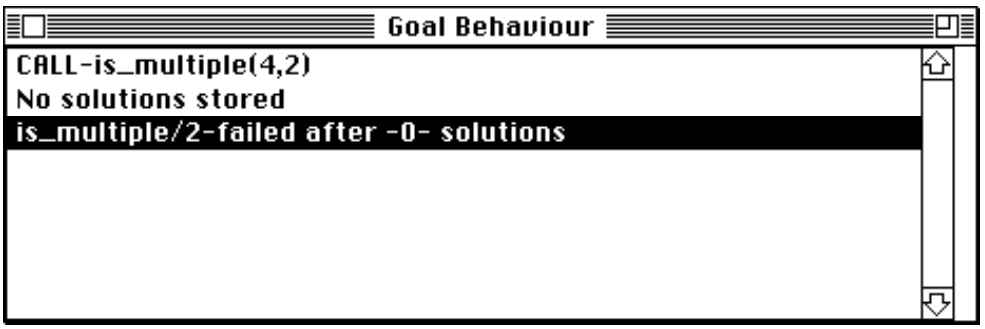

Figure 4.4: Showing a goal failure

After the user again invokes "WHY" on the selected failure, the diagnosis is presented:

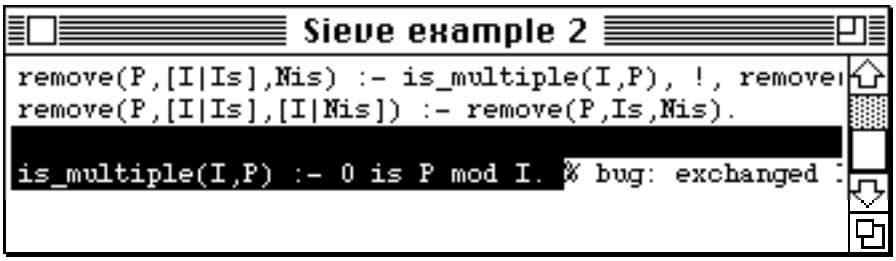

Figure 4.5: Showing an incomplete predicate

→

4.2. Output side-effects

Historically, logic programming has adopted a certain style of programming input/output interaction, typically through the use of destructible (i.e., non-backtrackable) actions on character streams, in the middle of normal logical deductions - *side-effects*. Although current programming practice is starting to avoid this approach, separating as clearly as possible interface from purely deductive code, the traditional approach is still common. Even the current drafts of the future Prolog standard continue to ignore this distinction, relegating the issue to each implementation.

For the sake of completeness, we'll expound a method to debug programs using output side-effects (such as `write`). In a subsequent section we'll discuss the serious problems involved in supporting also input side-effects (such as `read`). We'll tolerate input side-effects in a program, but we won't be able to find bugs related to their use.

We'll start defining a trace of output side-effects for each goal, to which we call the goal's segment. We then define additional types of oracle statements, and hence of bug manifestations, bug instances, etc. This approach was first presented in [Pereira, 1989 #118].

4.2.1. Extending the oracle

Let's define (output) *segment* of a goal to be the sequence of output side-effects (say, "writes") in the order they occurred in time, during that goal's execution and *under*[1] it. Notice that a segment of a goal may be chronologically interleaved with parts of segments of other goals.

Example Take the following Prolog program:

```
p :- write(1).    q :- write(3).
p :- write(2).    q :- write(4).

top :- p,q,fail.
```

The segment of `top` is [1,3,4,2,3,4]. The segment of p is [1,2]. Notice that the segment gives meaning to predicate definition `top`, which would be meaningless if we regarded this as a normal program, given that `fail` is always false. →

The goal behavior for a call G is now extended with G's segment as an *additional facet*, represented in the debugger theory as a fact:

• goal_segment(call(G),S).

S is a unique representation of the output segment, which the debugger may later use to display to the user. For example, a unique segment name plus a list of the names of output side-effect calls, ordered according to their temporal order. For convenience, we'll also use a functional notation to denote the name (unique among goal behavior facets) of a segment for a goal G:

• segment(G).

In addition to wrong solutions and incomplete solution sets, an additional bug manifestation now exists: *wrong output* segment. Let's revisit the table of possible oracle statements for each facet type, for programs with cuts, and add the additional case:

---

[1] As defined previously.

Type	Status	Meaning
solution	correct	solution is correct (wrt a particular goal call)
solution	incorrect	solution is incorrect (wrt a particular goal call)
solution set	correct	all solutions for the goal are correct (wrt it) and none is missing
solution set	incorrect	the solutions for the goal are correct (wrt it), but some are missing
segment	correct	goal produces correct output segment
segment	incorrect	goal produces wrong output segment

Table 4.3: Oracle framework for wrong output

Notice that although output side-effects are a non-declarative feature, statements about goal segments are declarative: they're valid irrespective of the context of the goal. That was the reason for defining segments wrt output *under* a goal, rather than *execution time*, for example.

Of course the notion of correct segment implies ordering among side-effects, which is dictated by a particular search strategy, such as Prolog's . For our purposes this issue is irrelevant: we assume that there's *some* ordering, of which the oracle is aware, but we need not know it.

The customized oracle theory will be the same as for programs with cuts, but with an additional rule for subsumed statements:

subsumed_facet(segment(G1),solution(G2))□
"Output of G1 identical to output of G2" □ "call(G1) identical to call(G2)"

4.2.2. An additional type of bug: wrong output

Whereas for cuts we just had to change existing definitions, for wrong output we must define an additional type of bug, using the additional type of goal behavior facet.

A *bad output predicate instance* is a predicate instance matching a goal with wrong output, but such that all goal calls in the predicate instance body have fully correct behaviors. The requirement for "fully correct goal behaviors" guarantees that a bad output predicate instance is not assumed so because of some erroneous subcomputation (be it a wrong or missing solution).

Example Consider the following buggy listing predicate, which is not inserting a newline at the beginning, but is instead inserting two between each `write_user` call:

```
buggy_listing(G) :-
    clause_user(G,B),
    nl_user, % Bug: should be before the previous goal
    write_user((G:-B)), nl_user,
    fail.
buggy_listing(G).
```

For goal `buggy_listing(remove(A,B,C))`, where `remove` is one of the predicates in the primes example in the previous section, the matching predicate definition is a bad output predicate instance, because although its output is incorrect (cf. its output segment below), all subgoals match correct system predicates.

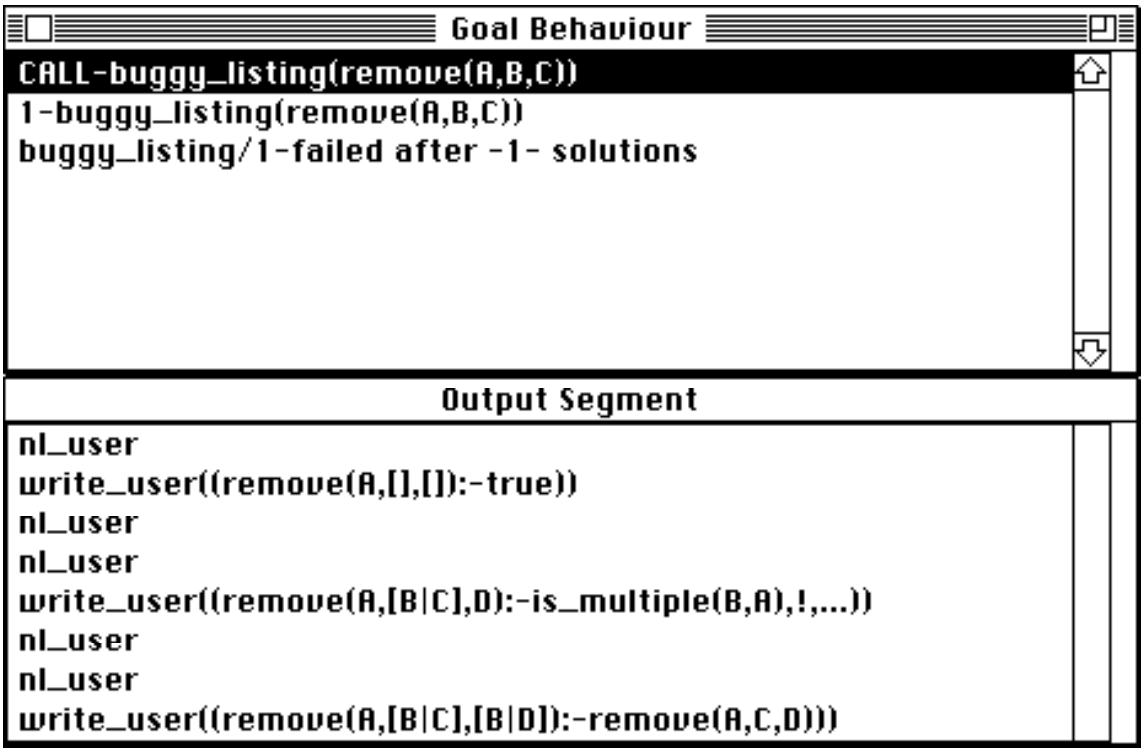

Figure 4.6: A bad output predicate instance

→

4.2.3. Debugging erroneous output

We'll now define a suspect tree for wrong output debugging. We'll simply define a suspect tree in the same format as the previous one for programs with cuts. It will be distinguished from that simply because the node names are disjoint: they're goal segment names, unique among *all* goal behavior facets (including solutions and solution sets).

The suspect tree for the output segment of a goal T, ST(segment(T)), is defined as follows:

• Each node has the name of a goal output segment facet, and is labelled with the predicate instance matching the goal.

• The root is the node segment(T).

• Let segment(G) be a node. Then for each goal call Gi in the predicate body instance for G, the node has a child ST(segment(Gi)).

Given this new type of suspect tree, syntactically similar to those for wrong solutions and incomplete solution sets, the debugging framework developed so far can be used almost “as is”. With just a slight difference, patent in the following proposition.

Proposition 10 Consider a goal G with wrong output, and that all goals in the computation for G have correct goal behavior facets, except for their segments whose status is undetermined. Then there is a bug instance in SS(segment(G)), a bad output predicate instance.

Proof From the definition of SS(segment(G)), and by induction on the level of ST(segment(G)). →

The only difference regarding side-effects-free programs is that debugging may start with a wrong output segment, but later may continue with a wrong solution, inadmissible goal call or incomplete solution set, if such are found by the oracle. Notice that additional information is required: whenever the oracle states a goal behavior to be correct, it must be aware that all its facets, including the segment, are correct.

Example Following is a buggy program to pretty print a term, by indenting subterms:

```
display_tree(T) :-
    nl_user,
```

```
    display_tree(T,0).

display_tree(T,Level) :- dt_leaf(T), !,
    dt_write_spaces(Level),
    write_user(T),
    nl_user.

display_tree(Tree,Level) :-
    Tree=..[F|Args],
    write_user(F),
    nl_user,
    L1 is Level+1,
    dt_member(T,Args),
    dt_write_spaces(Level), % bug: should be L1
    display_tree(T,L1),
    fail.
display_tree(_,_).

dt_member(T,[T|_]).
dt_member(T,[_|Trees]) :- dt_member(T,Trees).

dt_leaf(T) :- atom(T).
dt_leaf(T) :- integer(T).

dt_write_spaces(0).
dt_write_spaces(Level) :-
    Level >0, L1 is Level-1,
    write_user('--'),
    dt_write_spaces(L1).
```

Goal display_tree(a(b,c(d),e)) succeeds but produces wrong output:

```
a
b
c
--d
e
```

It should produce instead:

```
a
--b
--c
----d
--e
```

Using the HyperTracer, it is possible to complain about the segment of a goal call, by applying AD&Q with a menu command on a segment window:

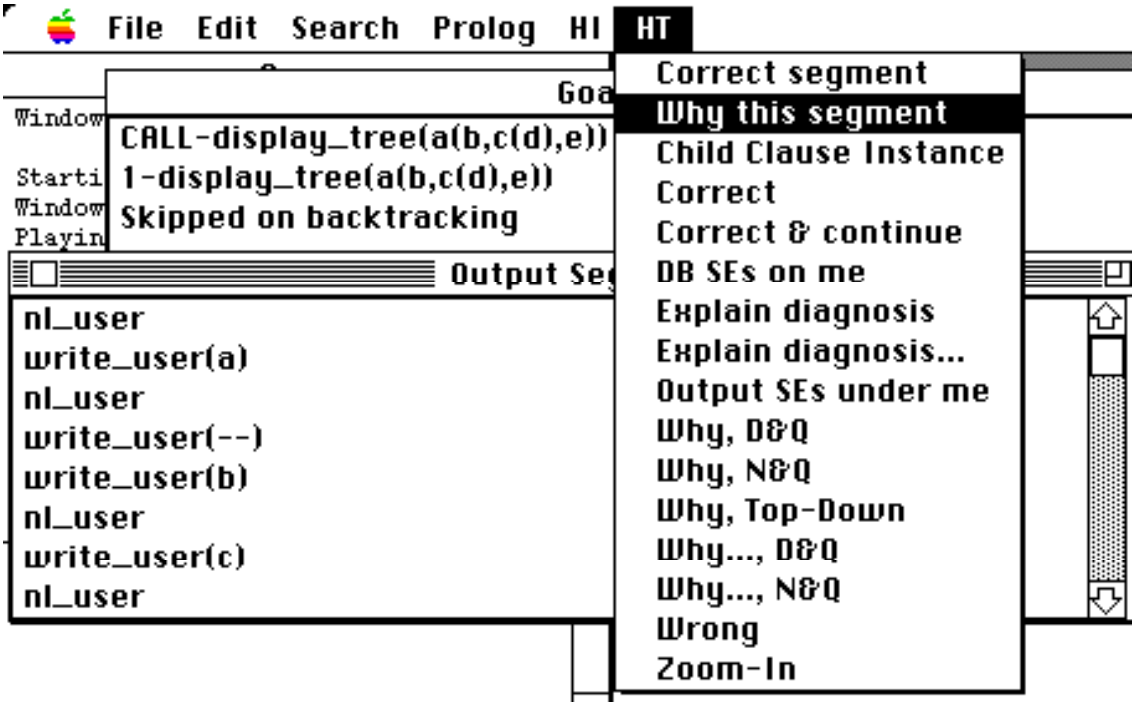

Figure 4.7: Complaining about wrong output

And again:

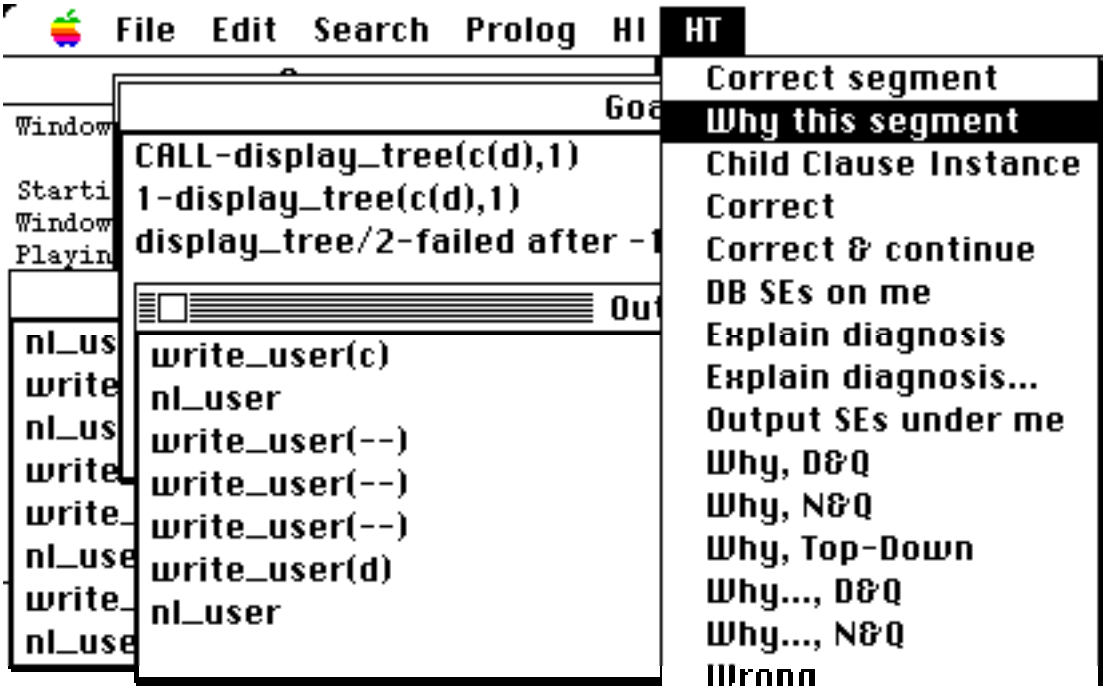

Figure 4.8: Debugging wrong output

Now a correct segment[1]:

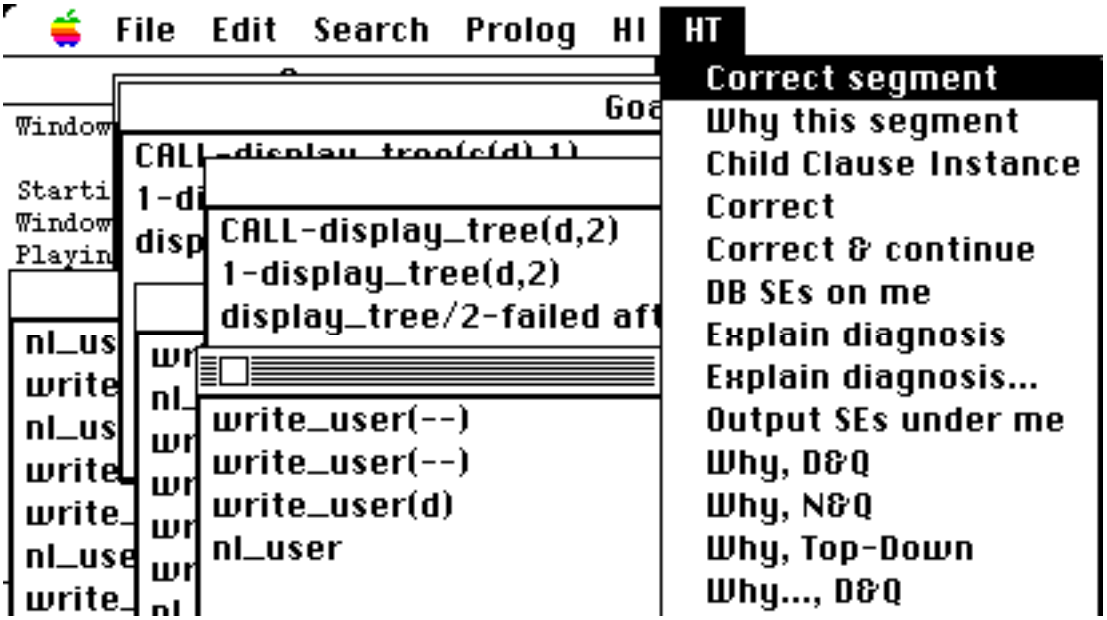

Figure 4.9: Debugging wrong output

dt_member(A,[d]) has a (correct) empty segment, and dt_write_spaces(1) also has a correct segment:

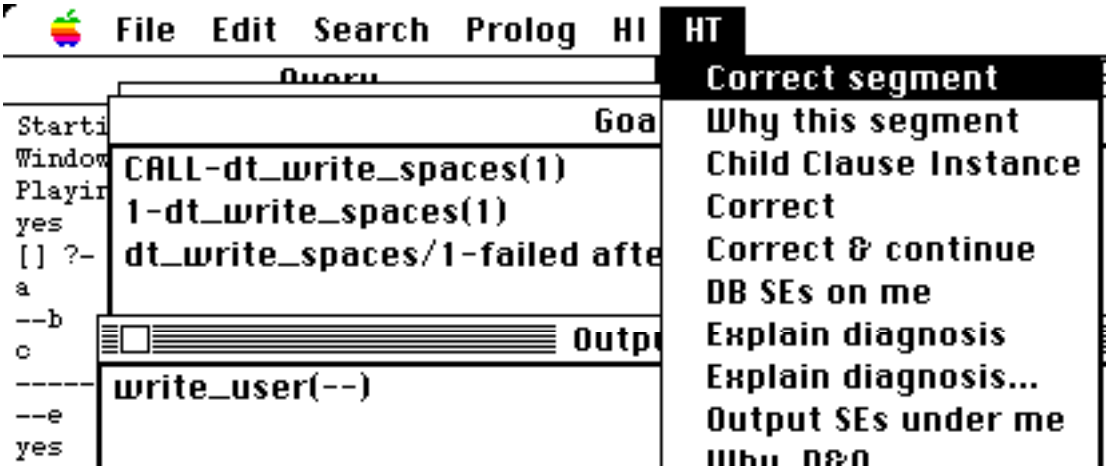

Figure 4.10: Debugging wrong output

[1] Whereas the HyperTracer fully supports AD&Q for wrong and missing solutions, and inadmissible goal calls, it currently requires the user to repeat the last "Why this segment" command (cf. "Correct and Continue" HyperTracer command description, chapter 6).

`dt_leaf(c(d))` fails, producing a correct and empty output segment. After this statement, diagnosis is reached, after a total of 6 queries: the predicate instance matching goal `display_tree(a(b,c(d),e),0)` is a bad output predicate instance. →

One last comment regarding the disadvantages of programs with side-effects. It will certainly become harder to apply the "Clever execution principle" put forward before (cf. chapter 2) to refine suspect sets without additional oracle knowledge. Different execution strategies, say different from Prolog's, will most likely violate the ordering of side-effects. This illustrates how a semantically misbehaved language feature is paid for with worse performance from a language environment tool - debuggers in our case, but also the case with partial evaluators for example ([81]).

4.2.4. Refined method

In [69] we introduced a refinement allowing diagnosis to start with a smaller suspect set, the "refined output suspect set", by using more detailed information from the oracle. In the present context, rather than defining an additional type of suspect set, this improvement can be seen as just another use of "intensional oracle statements" (cf. chapter 3).

Consider a wrong output segment for a goal G, S. Suppose that this "wrongness" can be pinpointed by the oracle by localizing a continuous nonempty (output) subsegment S', such that *no matter what the goal in the program for which S' is a subsegment, the goal's segment is wrong*. Such a statement can be codified into an oracle rule, allowing goal segments unseen by the oracle to be found wrong.

This defines side-effects which are erroneous "by themselves", irrespective of the goal call.

Example Consider the following program, with top goal `a`:

```
a :- b, c, write(1), e.
b.
c :- write(2), d, write(w).   % w should be a number
d.
e.
```

The initial suspect set would be RSS(segment(a), {o_s(wrong,segment(a)}) = {a/0, b/0, c/0, d/0, e/0}. But if the oracle theory OT includes instead a statement "any segment with non-integer output is wrong"[1], the refined suspect set would be RSS(segment(a), OT) = {c/0,d/0}. →

4.3. Other built-in predicates

A built-in predicate is a predicate definition provided by the logic program executor as a "system predicate" or "built-in", and therefore not explicit in the logic program itself. In addition to cuts and output side-effects, treated in the previous sections, we'll distinguish different cases according to the difference in character of built-ins.

4.3.1. The built-in can be seen as an implicit program predicate

This is the case for those builtins whose complete semantics can be described as an implicit program predicate definition, possibly with restrictions on its use given that typically such predicates are partially implemented. For example, arithmetic predicates are normally implemented as functions, and the user will get an error message if he tries to use them as full-fledged logical predicates: Y is 2*5+3 is properly evaluated, but 13 is 2*X+3 will cause an error message[2].

Examples of builtins in this category: arithmetic, `univ`, `call`, or in general any system predicate that could be simulated with an explicit program relation. Control meta-predicates, like Prolog's "if-then-else", `once`, etc. also fall into this category (they can be implemented with predicate definitions using the aforementioned builtins). Counter-examples: side-effects (assert, retract, input/output) are *not* in this category.

Debugging of programs with these features can be done by using our framework, by ommitting from suspect sets the (implicit) program components corresponding to the builtins.

Example Take the following program, with wrong solution p(1,1).

---

[1] Codified with something like "o_s(wrong,segment(G))<-goal_segment(G,S), ¬ all_integers(S)".

[2] Constraint logic programming systems [42] avoid this particular problem, but that's not the point.

```
(1)   p(X,Z):- Y is X+1, a_goal(Y,Z,G), call(G).
(2)   a_goal(A1,A2,g(A1,A2)) .
(3)   g(X,X).

(4)   is(2,1+1).  % implicit system predicates
(5)   call(g(X,Y)) :- g(X,Y).
      …
```

The suspect set for p(1,1) will be just {(1'), (2'), (3')} (i' meaning the computed instance of clause i). →

4.3.2. Built-ins accessing an "external world" with state

This is the case of those builtins whose complete semantics cannot be completely expressed as logical predicates. For example, builtins for file I/O: the result of a call to `get` or `read` is dependent on the previous calls affecting the particular file stream. Another example: a retract call, whose result depends on previous assert and retract calls for that particular predicate.

Although we treated output side-effects (cf. previous section), we have imposed the restriction that the external state change induced by output side-effects was irrelevant for the execution of the program. In other words, we assumed there was no causal connection between output side-effects and other predicates used by the program.

"Declarative debugging" of programs using (also) input side-effects is unelegant if not impossible. In order to classify goal behavior facets as correct or incorrect, the oracle would need to be aware of the subjacent *external state*, on which input side-effect calls depend.

Neverthless we'll concentrate on the specific case of program database side-effects (`assert`, `retract`, etc.), and come up with a partial solution, based only on declarative information.

4.3.3. Program database side-effects

We assume that buggy *internal* side-effects (assert, retract, and the like) have manifested themselves as some buggy top goal behavior. (Assuming otherwise would force us to extend our defined notion of "goal behavior" to include any goal execution's associated database state changes.)

Our basic idea is to extend the suspect sets, for those goal executions relying on predicate definitions modified by side-effects[1], with the suspects for changes: the suspect trees supporting the (potencially inadmissible) side-effect calls. Accordingly, we *extend* the previous definitions for suspect tree as below. This approach was first presented in [69].

First let us define the *sequence of changes to a predicate definition P* to be the sequence of database side-effect (`assert`, `retract`, `abolish`, ...) goal call facets directly changing P.

Example Consider the following program, and goal `top`.

```
top :- assert(p(1)), assert(q(1)), fail.
top :- assert(p(2)), retract(p(1)).
```

The sequence of changes for predicate p/1 is [assert(p(1)), assert(p(2)), retract(p(1))]. →

We'll now consider a time ordering between goal behavior facets, corresponding to execution time. Consider an SLDNF-tree, visited in some predetermined order along its top goal execution. A *side-effect goal call G potentially affects facet F* if the node G (i.e., where G is selected) is visited by the executor "before F", or more precisely before node F' as follows:

• If F is a goal call occurring in clause instance C, F' is the node immediately below the clause link labeled with C.

• If F is a goal solution, F' is the node after the first clause link in the solution path for F (i.e., the link labeled with the clause instance matching F) .

• If F is a solution set for goal G_F, F' is the last[2] goal call under G_F; in the special case where F' is G, G is still considered to be before F.

• If F is a segment, the same as for solution sets.

In other words, a side-effect potentially affects a goal behavior facet if it is executed *before* the interpreter uses the program component matching or containing the facet.

---

[1]The assert and retract calls by themselves aren't problematic: their side-effects on *other* goals are.

[2] I.e., such that there's no other facet before it under G_F.

We now present the *extended suspect tree*, in the form of a generic extension to the previous suspect tree definition (cf.previous section):

• Let F be a goal behavior facet node in a suspect tree, matching (or occurring in, for the case of calls) predicate P, a predicate with a nonempty sequence of changes. The node has an additional child ST(call(G)) for each side-effect call G *potentially affecting* F (cf. above) and in the sequence of changes to P.

For a goal execution which does not use predicates changed by assert/retract, the extended suspect tree simplifies to the nonextended one.

This extension brings an interesting novelty: suspect sets for a bug manifestation in a top goal may include goals in *other (previous) top goals*.

Example Consider the program

```
(1)   a :- b, asserta(h:-c,!,d).
(2)   h :- e.
(3)   e.
(4)   b.
```

Consider that goal `a` has been executed successfully, and that afterwards top goal `h` produces a wrong solution using clause (2). Then the suspect set is SS(h) = {(1), (2), (3), (4)}, and not just {(2),(3)}. →

Suspect sets and debugging algorithms can simply be redefined based on the new suspect trees.There are two subtleties, however:

• A (declarative) query about the admissibility of a call producing the asserting or retracting of a clause merely concerns its potential *type-violating* character. I.e. the user may notice a type violating subgoal in its body for example, but otherwise one cannot assume the clause to be “admissible” (in the sense of trusting there are no bug instances among the suspects supporting its assertion).

• Furthermore, this approach does *not* take into account any relative *order* of side-effects, as we don't find it reasonable to presuppose the user aware of that order.

Consequently, it is occasionally possible to obtain a vacuous diagnosis[1]. In any case, this approach seems an improvement over conventional tracing methods.

[1]As an example, consider a buggy predicate, whose implementation requires asserting terms in a particular order. When later diagnosing the troubles it caused, the query to the oracle about the admissibility of the assert calls does *not* take into account their ordering: the oracle will find the assert calls "admissible", so the debugger gets stuck with an identified wrong dynamically asserted clause.

5. Declarative Source Debugging

We've been concerned with finding a bug instance, given an initial bug manifestation. Let's consider instead the problem of finding **bugs** in the source program, rather than bug instances in the suspect tree, leading to *Declarative Source Debugging*. As will be seen, in general this allows a diagnosis to be found with less queries to the oracle.

The present chapter will explore this approach[1], and combine it with the debugging framework developed so far. As a result we come up with declarative source debuggging algorithms, for all bug types and language features considered in the previous chapters.

In order to do it, we'll simply adapt our previous framework, from now on refered to as “Declarative Execution Debugging”. The adaptation consists in a few additional definitions, and changes to the previous algorithms. However we will *not* require the use of different types of oracle statements.

In addition to optimum algorithms, minimizing the maximum number of queries in any circumstance, we'll present two algorithms with an heuristic component, suggested by the new focus on program source rather than on program execution: the Narrow&Query algorithm, which attempts to reduce the number of suspects as much as possible with each single query; and the SECURE algorithm, which starts assuming that program components contributing to a correct result are always correct, and later lifts this assumption if proven inadequate.

---

[1] We first mentioned the opportunity for declarative source debugging in [68], and later explored it in [14] and presented it at the Workshop on Programming Environments, before ICLP'91, in Paris.

5.1. Motivation

We start with some motivation for declarative source debugging.

Example Follows a buggy "append" predicate with a wrong solution `my_append([1,2,3,4,5],[6],[1,2,3,4,5,6,7,8])`:

```
(c1)  my_append([],L,K).      % bug: K≠L
(c2)  my_append([X|A],B,[X|C]) :- my_append(A,B,C).
```

The and tree of this solution is (the labels on the left denoting the source clauses that matched the goals):

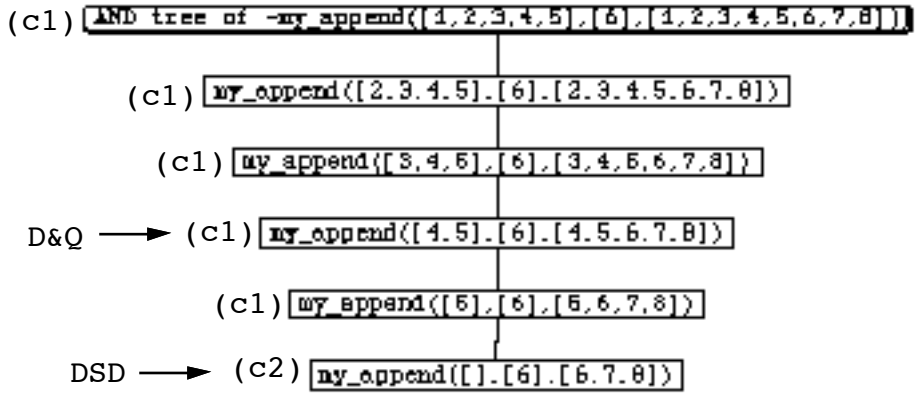

Figure 5.1: A suspect tree for `append`

The Divide and Query algorithm (or even GD&Q(∞)) would pick the middle goal node for querying the oracle, plus an aditional query or two until finding the bug. →

Now, notice that under node `my_append([],[6],[6,7,8])` there are only instances of clauses (one in fact: c2) which have no instances not under the node: not under the node there are only instances of clause c1. A ***declarative source debugger*** should query about that node first. Were it to be stated correct by the oracle, there would be a bug instance higher in the tree, in a region containing only instances of clause c1, and *no additional queries would be needed*; being wrong (as in fact it is), then clause c2 has a bug instance, and again *no additional queries are needed*.

Therefore, for this example and in the worst case, 2 queries are needed for a "declarative source debugger", rather than the 4 for a "conventional declarative debugger". (We count the initial incorrectness statement for the top node as a query.)

This example suggests that while a conventional "declarative execution debugger" can find a bug instance with a number of queries logarithmic with the number of AND tree nodes, a declarative source debugger might be able to find a bug with a number of queries dependent on the number of source *clauses* in the program - a much more interesting prospect, since logic programs are apt to use recursion.

5.2. Declarative source debugging defined

We start by defining the "source suspect set", SSS, from our previous suspect set SS. The *source suspect set* for a goal behavior facet F, *SSS(F)*, is the set of all program components with computed instances in SS(F).

Example The source suspect set for wrong solution `my_append([1,2,3,4,5],[6],[1,2,3,4,5,6,7,8])` in the previous example is simply **{c1,c2}**; SS would be:

```
{
my_append([1|[2,3,4,5]],[6],[1|[2,3,4,5,6,7,8]]):-
       my_append([2,3,4,5],[6],[2,3,4,5,6,7,8]),
my_append([2|[3,4,5]],[6],[2|[3,4,5,6,7,8]]):-
       my_append([3,4,5],[6],[3,4,5,6,7,8]),
...,
my_append([],[6],[6,7,8]):-true
}
```

→

Example Let's revisit the `display_tree` example of the previous chapter. The suspect tree for the wrong output segment of goal `display_tree(a(b,c(d),e))` was (although not shown) the following, with a total of 29 suspect nodes:

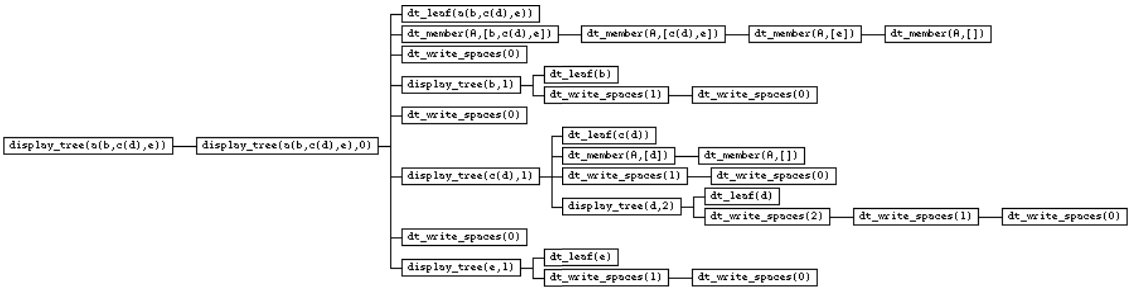

Figure 5.2: A suspect tree

The source suspect set would simply be the set of 6 predicate definitions in the program ({`display_tree/1, display_tree/2, dt_member/2, dt_leaf/1, dt_write_spaces/1`}). →

We can easily adapt previous propositions for suspect sets:

Proposition 11 Given a bug manifestation T, its source suspect set SSS(T) contains a bug.

Proof SS(T) contains a bug instance I, by proposition 1. Then the program component (bug) of which I is a computed instance is in SSS(T). →

Unfortunately we can*not* adapt proposition 3 directly: given a bug manifestation F and a node F' in ST(F) for which o_s(correct,F'), in general we can't ignore all program components in SSS(F'), because they may have suspect instances not in ST(F').

But we can define “refined source suspect sets” indirectly. Given an oracle theory OT, and a bug manifestation F:

A *refined source suspect set*, RSSS(F,OT), is the set of all program components with computed instances in RSS(F,OT).

Proposition 12 Given a bug manifestation F and an oracle theory OT, RSSS(F,OT) contains a bug.

Proof Follows directly from proposition 1 and the last definition.→

5.3. Finding bugs

5.3.1. Bug instance search trees revisited

From a bug instance search tree (BIST) we can now define the corresponding “bug search tree”.

The *bug search tree* for a bug manifestation T, given an oracle theory OT, is denoted by BST(T,OT), and is obtained from BIST(T,OT) as follows:

- Consider a debugger node bist(F,OT'). If one of the sets RSSS(F,OT') is a singleton, delete all descendents of the node.

In other words, whenever all suspects are instances of the same program component, make the node a leaf. Apart from this difference, BISTs and BSTs are identical.

Example Following is a snapshot of BIST(solution(`my_append( [1,2,3,4,5], [6], [1,2,3,4,5,6,7,8] )`)), showing the groups of nodes which collapse into a single BST leaf node. Notice how the branch after the lower query node (corresponding to the suspect node at the bottom of the `my_append` recursion chain) collapses into a single node, thus ponting out that suspect as the best to query about first.

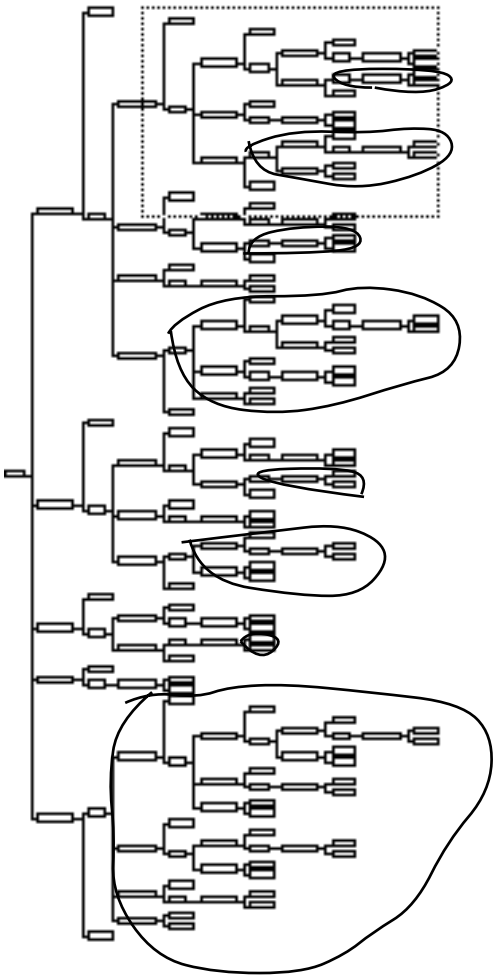

Figure 5.3: A Bug Search Tree

The rectangle at the top can be seen in more detail below (app1 abbreviating my_append(...) nodes matching clause c1, and app2 those matching clause c2):

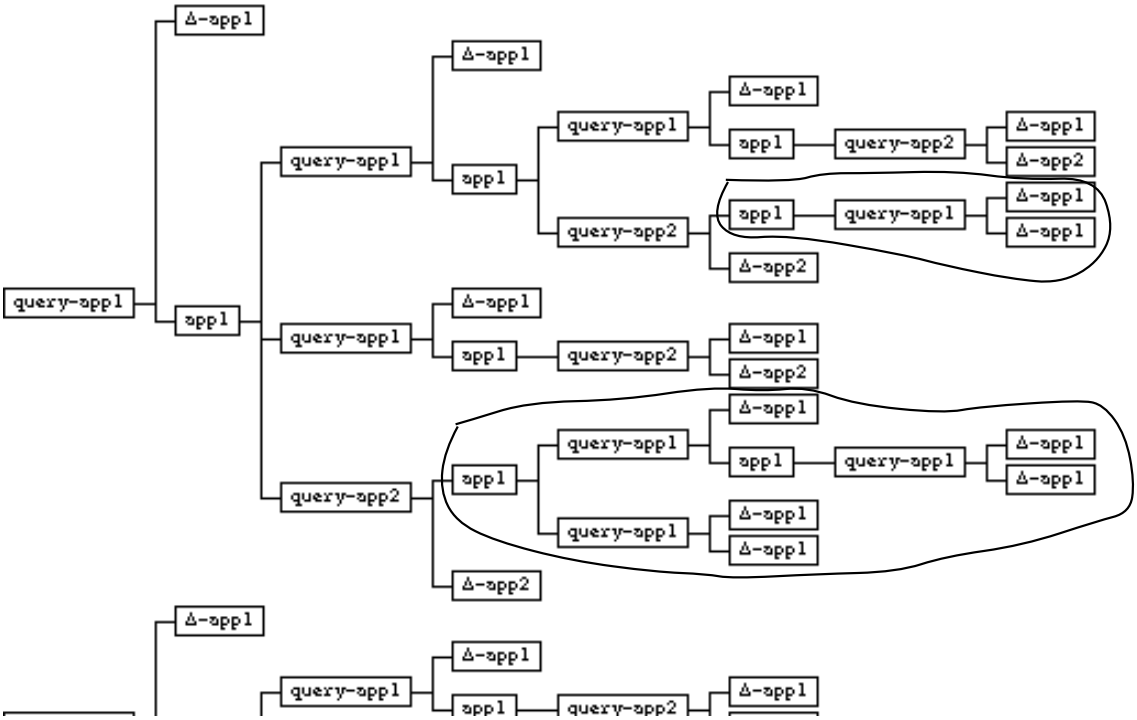

Figure 5.4: Detail of a BST

→

The next propositions sound familiar:

Proposition 13 Given a bug manifestation T and an oracle theory OT, a program component is a bug if a computed instance of it matches the goal behavior facet in a leave of BST(T,OT).

Proof All leaves are debugger nodes, because bug nodes always have at least one child. If a debugger node bist(F,OT') has no children, then (one of the sets) RSSS(F,OT') has a single element, the program component C whose instance matches F. Since F is a bug manifestation, by definition of debugger node, then by proposition 12 C is a bug.→

Proposition 14 Given a bug manifestation T and an oracle theory OT, there are no certified bugs with computed instances matching goal behavior facets in non-leave nodes of BST(T,OT).

Proof Take a debugger node bist(F,OT') with descendents. Since the node is not a leaf, #RSSS(F,OT') > 1 for all RSSS sets: there is more than one suspect, hence we can't arbitrate one of them to be the diagnosis. →

5.3.2. The basic algorithm

From the basic declarative execution debugging algorithm, that uses the bug instance search tree, we can define a new basic algorithm for declarative source debugging, by using the *bug search tree* instead. Notice that only steps 1 and 2 are different.

1. Given a bug manifestation T and oracle theory OT, compute BST(T,OT). Make the root the current node.

2. If the current node is a leave bist(G,OT'), return as diagnosis the single element of any RSSS(G,OT') which is a singleton.

3. Select some child of the current node, a bug node query(Fi,OTi).

4. Query the user about the correctness status S of goal behavior facet Fi ("correct or incorrect ?") and continue at step 2 with bist(Fi, OT ≈ u_s(S,Fi)) as the current node.

Just as the original declarative execution debugging that searches bug instances, the present algorithm should not look very "intelligent" to a user, because it typically makes much more queries than necessary. But it serves as a conceptual basis for the following improved ones.

5.3.3. "Intelligent" algorithm

The discussion and changes to the basic declarative execution debugging algorithm apply directly - only step 3 is redefined:

3'. Select a child of the current node, a bug node query(Fi,OTi), such that the *maximum cost* for any diagnosis query sequence starting in the node is minimum.

5.4. Search heuristics

While the number of execution suspects, #RSS, was a diagnosis cost bound, #RSSS is not. The query cost continues to depend on the execution's suspect tree: clearing a source suspect instance does not necessarily clear all instances of that source suspect.

We now define "collapsed" suspect tree, taking into account the relationship between suspect tree nodes and source program components. This will suggest a lower diagnosis cost bound, to be used as a search heuristic.

The *collapsed suspect tree* for a goal behavior facet F, CST(F), is obtained from the suspect tree ST(F) as follows:

- Consider all trees T_C in ST(F), with root C', and such that all nodes in T_C match the same program component C.
- Let b be the maximum branching of ST(F). Replace T_C by a single node C', with a child for each node that is an immediate descendent of a node in T_C, except in the following case.
- If C' has more than b children, remove enough instances of C from T_C such that the number of children of C' becomes ≤b.

It is easy to see that if SSS(F) contains a bug, the set of program components with instances in CST(F) also does: the "collapsed" nodes are additional instances of program components, irrelevant from the point of view of finding bugs.

Example Let's revisit the buggy prime number program, giving names to program components, clauses (ci) and predicate definitions (pdi):

```
(pdA)
(c1) primes(Limit,Ps) :- integers(2,Limit,Is), sift(Is,Ps).

(pdB)
(c2) integers(Low,High,[Low|Rest]) :- Low =< High, !,
       M is Low+1, integers(M,High,Rest).
(c3) integers(_,_,[]).

(pdC)
(c4) sift([],[]).
(c5) sift([I|Is],[I|Ps]) :- remove(I,Is,New), sift(New,Ps).

(pdD)
(c6) remove(P,[],[]).
(c7) remove(P,[I|Is],Nis) :- is_multiple(I,P), !, remove(P,Is,Nis).
(c8) remove(P,[I|Is],[I|Nis]) :- remove(P,Is,Nis).

(pdE)
(c9) is_multiple(I,P) :- 0 is P mod I. % bug: exchanged I,P
```

Following is ST(solution(primes(4,[2,3,4]))), considering the effect of cuts, and showing the names of the source program components matching suspects. The selected nodes at the left are the only ones collapsing into a single node in CST(solution(primes(4,[2,3,4]))); although other trees of nodes matching the same program component exist, their collapse would cause the overall maximum tree branching factor to increase, hence violating the CST definition.

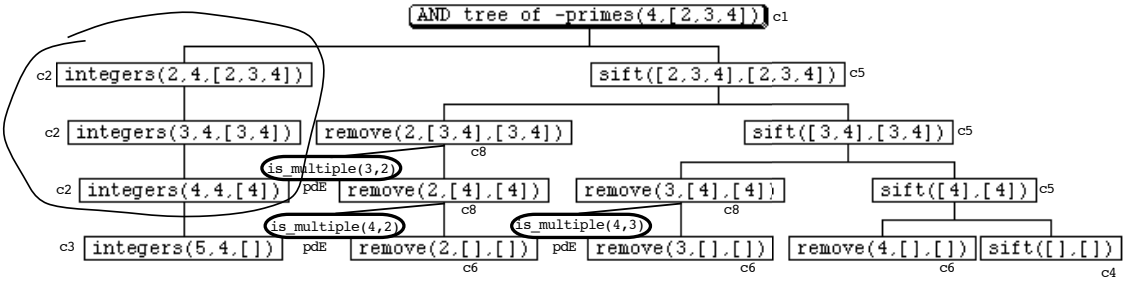

Figure 5.5: A Collapsed Suspect Tree

→

Given an oracle theory OT, and a bug manifestation F, the *collapsed refined suspect tree*, CRST(F,OT), is defined as CST(F), but based on RST(F,OT) instead of on ST(F) .

A better upper bound on the number of queries can now be established. Given a bug manifestation T and an oracle theory OT, the *form-aware bound on source diagnosis cost*, $h_{fs}(T,OT)$, is defined as follows: the smallest value $b \log_b n$ (for all trees CRST(T,OT)), where b is the maximum number of children of any node in CRST(T,OT), and n is the number of nodes in the tree.

Proposition 15 Let T be a bug manifestation, OT an oracle theory, and bist(T,OT) the root node of BST(T,OT). The maximum diagnosis query cost[1] for a bug in BST(T,OT) is less or equal to $h_{fs}(T,OT)$.

Proof By applying the Divide and Query algorithm on the collapsed suspect tree, a bug instance will be found with $b \log_b n$ queries, for the syntactically identical case where the tree is an AND-tree. The only difference lies in the nature of the diagnosis: in addition to a bug instance, it may also consist in a "collapsed" subtree, but in any case uniquely identifying a buggy program component. →

[1] I.e., following a strategy that attempts to minimize cost.

We now define a new algorithm, *Source Divide & Query (N)*, or *SD&Q(N)*, using this upper bound and inspired on the basic declarative source debugging algorithm above. It is similar to Generalized Divide & Query(N), except for the use of bug search trees and collapsed suspect trees:

> 3''. Consider the set S_N of debugger node descendents bst(F,OT) of the current node, that are N plies (2N levels) below in the tree, or higher up if they're leaves. Select a child of the current node, a bug node Q, minimizing the maximum h_{fs}(F,OT) for any descendent of Q in S_N.

Assuming that bugs are as interesting a diagnosis as bug instances, how better is SD&Q(N) than GD&Q(N) ?

The answer depends on the program and computation at hand. First, *it is never worse*. If no “suspect collapsing” is permited, because instances of program components are scattered around the suspect tree, *SD&Q(N) degrades to GD&Q(N)*. If instances of program components are concentrated in regions of the suspect tree, SD&Q(N) is manifestly better. But if instances of a program component are concentrated in a region of the suspect tree, but are interleaved with another's, the advantage vanishes.

We didn't investigate whether program properties definable from static analysis may help quantify the query performance.

5.5. Domain heuristics

Just like for declarative execution debugging, and in addition to search-related heuristics, we may use information about the probability of an user answer. If, for a goal behavior facet, we can assign different probabilities for the user stating it correct or incorrect, we can adapt SD&Q(N) accordingly, to minimize the *expected* diagnosis cost. To the result we call *Probabilistic Source Divide and Query(N)*, or *PSD&Q(N)*:

3'''. Consider the set S_N of debugger node descendents bst(F,OT) of the current node, that are N plies (2N levels) below in the tree, or higher up if they're leaves. Select a child of the current node, a bug node Q, minimizing the expected[1] h_{fs}(F,OT) for any descendent of Q in S_N.

5.6. Source-directed diagnosis

The previous algorithms used no domain information at all: they followed directly from the redefinition of the debugging problem - find a bug, rather than a bug instance. We now develop less conservative algorithms, which are based on additional assumptions involving the relationship between source program and computation.

5.6.1. Narrow & Query

Since debugger users are typically impacient, providing as much feedback as possible to them is important. An expert user is likely to skip the remaining of a diagnosis session, if he's given interesting information from which he can jump quickly to some conclusion.

Whereas in declarative execution debugging the suspect sets contain computed program component instances, in declarative source debugging suspect sets contain program components. This suggests that *a debugger can give feedback to the user by presenting him the current suspect set*, in the original source form.

We now present an algorithm for impacient users, based on these ideas. It is akeen to the Abstract Divide &Query algorithm, except that it “narrows” sets of program components, instead of “dividing” suspect trees.

The *Narrow & Query* algorithm, or N&Q, is defined from the basic declarative source debugging algorithm, as follows:

3'''. Select a child of the current (debugger) node, a bug node query(Fi,OTi) with two (debugger node) children bist(A,OT_A) and bist(B,OT_B), such that max{ #RSSS$_{SMALL}$(A, OT_A), #RSSS$_{SMALL}$(B, OT_B) } is minimum.

---

[1] By taking a weighted average of the estimative function in all nodes N plies below (or higher if leaves), with the combined probabilities of the query/answer sequences until them.

Given an oracle theory OT and a bug manifestation F, a *smallest refined source suspect set*, $RSSS_{SMALL}(F,OT)$, is any of the RSSS(F,OT) sets minimal regarding inclusion.

The algorithm chooses a query which minimizes the next source suspect set, irrespective of it being a good choice or not in the long term, when the total cost of additional queries is accounted for.

Example Following is a buggy partition predicate, producing wrong solution partition([1,5,3,4,2], 3, [1,3,2], [x,x]):

```
(c1) partition([X|A],Y,B,[x|C]) :- Y<X, partition(A,Y,B,C). % bug:x
(c2) partition([X|A],Y,[X|B],C) :- Y>=X, partition(A,Y,B,C).
(c3) partition([],_,[],[]).
```

And here's ST(solution(partition([1,5,3,4,2], 3, [1,3,2], [x,x]))). The numbered arrows on the left show the sequence of queries until diagnosis with Narrow&Query; the nodes marked with (AD&Q) constitute the query sequence for AD&Q (there's no improvement with GD&Q(N)).

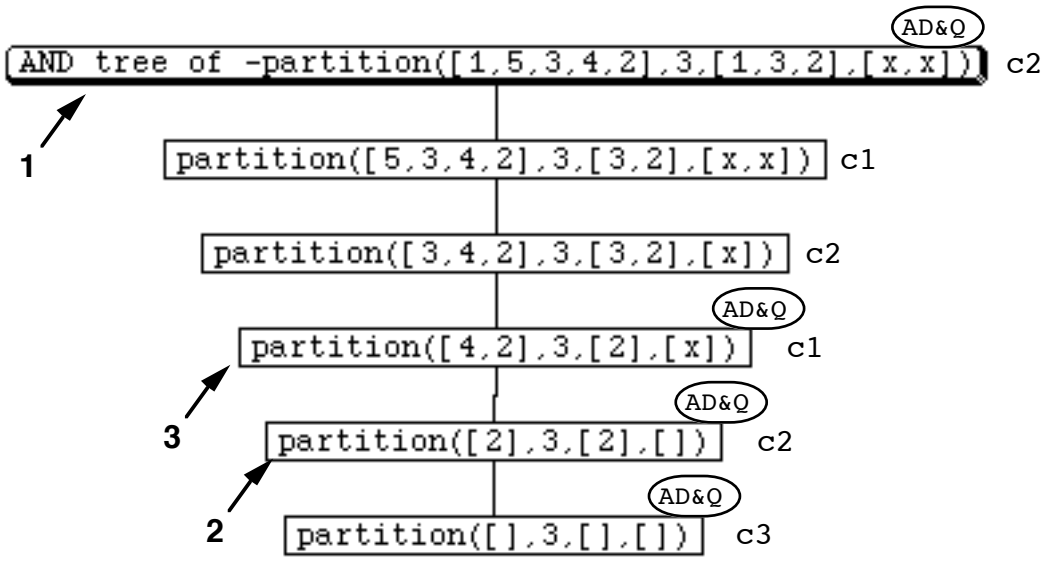

Figure 5.6: Debugging with Narrow&Query

The first query after the top goal solution, about (correct) solution partition([2],3,[2],[]), is chosen because it is one of the two queries minimizing the cardinality of the remaining source suspect set, 2 at most.

The "short-sighted" nature of this algorithm, and its eagerness to immediately minimize source suspect set cardinality, are manifest in this example. The other possible first query would be about (correct) solution partition([],3,[],[]), also guaranteeing a remaining cardinality of 2. However, the suspect tree would be such that it would be impossible to guarantee that the number of source suspects decreased, because of the interleaving of nodes matching c1 and c2:

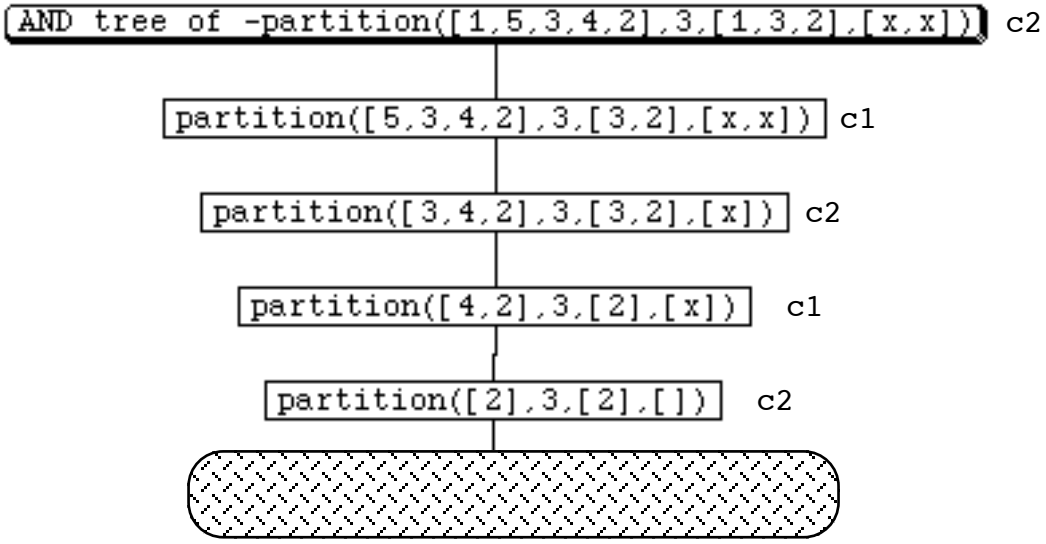

Figure 5.7: A case against Narrow&Query

→

So although each step of Narrow & Query may give useful information to a programmer, say by selecting the program source to point the current suspect set, another approach is needed for guaranteeing that a diagnosis can be found with advantage regarding declarative execution debugging algorithms.

5.6.2. The SECURE algorithm

We now move to another algorithm, which attempts to balance the use of source information, the use of a domain-dependent but intuitive assumption about bug instances, and some consideration for the worst case situations where declarative source debugging degrades into declarative execution debugging.

The algorithm is based on the idea of adopting an assumption until reaching an "assumption diagnosis" which is validated, or otherwise retracting the assumption and falling back into Generalized Divide and Query(N). For this reason it is called SECURE(N). We call "SECURE" to algorithm SECURE(1).

The SECURE assumption is the following. Let F be a correct goal behavior facet; then the program components in SSS(F) are assumed correct, and are omited from all current source suspect sets used by the debugger. In other words, a program component with a computed instance under a correct goal behavior facet is assumed correct.

Let ACC(T) be the set of Assumed Correct Components given oracle theory T.

SECURE(N) follows:

1. Given a bug manifestation T and oracle theory OT, consider BST(T,OT) and its root node bist(T,OT).

2. If the T node is a leave, return as diagnosis the program component in any singleton RSSS(T,OT).

3.1 **If** ACC(OT)∈RSSS(T,OT), **then** select a child query(Fi,OTi) of the T node according to step 3 of GD&Q(N) (cf. section "Suspect tree form factors" in chapter 2), with bist(T,OT) as current node and ignoring the SECURE assumption; proceed at step 4 below.

3.2 **else if** #[RSSS(T,OT)\ACC]=1 for some RSSS **then**

Let D be the (only) program component in RSSS(T,OT)\ACC; D is the "assumption diagnosis". Now attempt to validate it:

Find the instance DI of D in one of the smallest RSS(T,OT) sets for which there are more statements by the oracle (be it about the predicate head or body goals); if there's more than one such instance, matched by goal behavior facet F, choose the one with the smaller RSS(F,OT) - because we expect F to be incorrect.

Query about the remaining parts of DI, until one of the following conditions arises:

- if DI is a bug instance: terminate returning DI as the diagnosis;

- if a new bug manifestation B was found among the goals in DI's body: continue at step 1 with T=B.

3.3 **else** (*RSSS(G,OT')\ACC has always more than one element*)

Select a child query(Fi,OTi) of the T node, minimizing | #[RSSS(Fi, OTi)\ACC] - #[RSSS(T, OT)\ACC] / 2 |, considering any of the smallest RSSS sets; if this criterium selects more than one node, choose *from these* the best according to step 3 of GD&Q(N), with bist(T,OT) as current node and ignoring the SECURE assumption.

4. Query the user about the correctness of goal behavior facet Fi, add the answer to the oracle theory OT as a user statement, obtaining OT', and continue at step 2 with bist(Fi,OT') as the current node.

Like the previous algorithms, SECURE descends through the bug search tree until a diagnosis is found.

Normally SECURE returns a bug diagnosis using step 2, eventually after validating it (step 3.2). But, for impacient users, SECURE can optionally skip the validation step, and provide immediate feedback by presenting the assumed diagnosis.

We now show the need for the query overhead of the validation step (3.2):

Example

```
r(X,Y) :- p(X), p(Y).
p(X):-q(X). % Bug: clause is too general
q(a).       q(b).
```

Take the intended model to be {r(a,a), p(a), q(a), q(b)}. Therefore, for wrong solution r(a,b) there's a bug instance p(b):-q(b), an instance of the second source clause. Now, when the oracle states solution p(a) to be correct, SECURE will assume the second clause to be correct. Its diagnosis validation stage checks the validity of the assumption, and in this case retracts it, falling back on GD&Q (step 3.1).

5.6.3. The SECURE assumption

How good is the SECURE assumption ? It is inspired in an heuristic used by human debuggers: if a correct computation uses a program component, it is considered correct. Given the difficulty in analysing objective pros and cons, due to the diversity of logic programs, we experimented with SECURE to see if the assumption diagnosis are valid.

Following is a table with the maximum number of queries necessary until diagnosis, for each of the examples (taken from [14], where all listings and tree figures can be found). The initial bug manifestation is counted as a debugger query.

Example	#SS(Top)	#SSS(Top)	AD&Q queries	SECURE, part I	Validation
Partition[1]	6	3	4	2	1
Insertion sort[2]	17	5	5	3	0
Quicksort	26	7	6	4	2
Simple rule engine[3]	8	4	4	3	0
Sieve of Erastosthenes[4]	19	8	6	4	0

Table 5.1: Evaluating SECURE

The first column gives the number of nodes in the suspect execution tree for the wrong solution, and the second the number of source clauses matching goals in it. These are the initial numbers of suspects for declarative execution debugging and for declarative source debugging, respectively. The "SECURE, part I" column contains the number of queries until obtaining an assumption diagnosis.

The rightmost column contains the additional diagnosis validation queries for SECURE. In a practical programming environment, as soon as the debugger has a diagnosis to be validated it can display it to the user, so that the user can decide on whether to accept the diagnosis without validation or to answer additional queries.

In this set of examples the SECURE assumption was always successfully validated.

Example Here's the quicksort program referred to in the table above:

```
(c1)  qsort([],[]).
(c2)  qsort([X|L],L0) :-
            partition(L,X,L1,L2),
            qsort(L1,L3),
```

[1] Cf. previous example in the "Narrow & Query" section.

[2] From [83], p.50

[3] From [7].

[4] From [4].

```
            qsort(L2,L4),
            qappend([X|L3],L4,L0).  % bug: X should be "inserted" in the 2nd
arg.
(c3)   partition([X|L],Y,[X|L1],L2) :- X<Y, partition(L,Y,L1,L2).
(c4)   partition([X|L],Y,L1,[X|L2]) :- Y=<X, partition(L,Y,L1,L2).
(c5)   partition([],X,[],[]).
(c6)   qappend([X|L1],L2,[X|L3]) :- qappend(L1,L2,L3).
(c7)   qappend([],L,L).
```

And here's the suspect tree for wrong solution `qsort([2,3,1,5],[2,1,3,5])`, with numbered arrows pointing the SECURE query sequence, and D&Q the Divide&Query (or AD&Q, in this cut and negation-free program) queries.

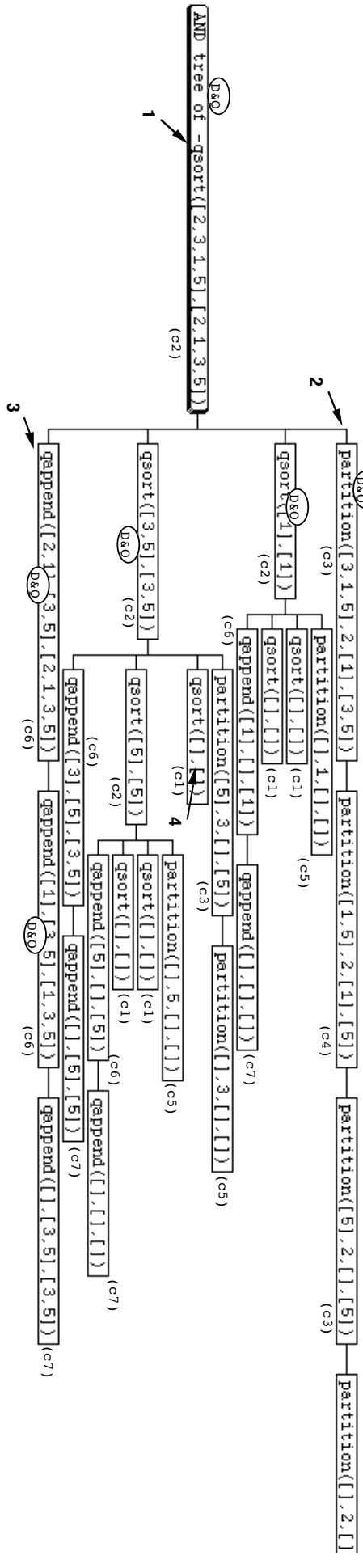
AND tree of -qsort([2,3,1,5],[2,1,3,5])
D&O
1
(c2)
2
3
4
qsort([3,1,5],2,[1],[3,5])
(c3)
qsort([1],[1])
(c2)
qsort([3,5],[3,5])
(c2)
qsort([],[4])
(c1)
partition([3,1,5],2,[1],[3,5])
partition([],1,[],[])
(c5)
partition([1,5],2,[1],[5])
(c4)
partition([5],2,[],[5])
(c3)
partition([],2,[]
qsort([],[])
(c1)
qappend([1],[],[1])
(c6)
partition([5],3,[],[5])
(c3)
qappend([],[],[])
(c7)
partition([],3,[],[])
(c5)
qsort([5],[5])
(c2)
qappend([2,1],[3,5],[2,1,3,5])
(c6)
qappend([3],[5],[3,5])
(c6)
qappend([1],[3,5],[1,3,5])
(c6)
qappend([],[5],[5])
(c7)
qappend([],[3,5],[3,5])
(c7)
qsort([],[])
(c1)
qappend([5],[],[5])
(c6)
qappend([],[],[])
(c7)
partition([],5,[],[])
(c5)

Figure 5.8: Example of AD&Q vs SECURE

→

5.7. Complexity issues

Source Divide&Query(1) needs at most as many queries as GD&Q(1), when program component instances are scattered around the (execution) suspect tree: b $\log_b$ n, where n is the number of execution suspect tree nodes and b the maximum branching factor of the tree.

But if instances of the same component are grouped together in (disjoint) suspect tree regions, the number of queries will be dramatically less: it will be that of GD&Q(1) over the *collapsed* suspect tree. If C is the number of program components in the source suspect set, the number of queries will be O(log C), rather than O(log n).

The Narrow&Query algorithm "hopes" that program component instances are grouped together, and simply tries to elliminate as suspects half of the current source suspects; it corresponds to Abstract Divide & Query[1] over the collapsed suspect tree, and therefore will also be O(log C) in the number of queries, eventually with a small query overhead regarding SD&Q because it ignores tree form factors. If components are not grouped together, the query performance may degrade to O(n).

The SECURE algorithm adopts an assumption which is equivalent in practice to having program component instances grouped together. Therefore it is able to eliminate about half of the source suspects in a single step like Narrow&Query, if the assumption is valid. So in the worst case, when the assumption is found invalid and SECURE falls back into GD&Q, the number of queries will be[2] O(log C+log (n-C)).

---

[1] i.e., GD&Q(1) ignoring tree form factors.

[2] Any query done before the SECURE assumption is found invalid refines at least one execution suspect, hence the component log(n-C).

Part II: Implementation

6. The HyperTracer environment

We've implemented several experimental debugger prototypes, incorporating our evolving theory. In this chapter we describe our latest and most complete[1], the HyperTracer[2].

Before we do so, we summarize part I of this thesis, to better see where the prototype stands. We then state our design objectives, and say what's left out of the current implementation. An overview of the HyperTracer architecture, an example session and a description of the available commands follow.

---

[1]Running over C-Prolog, using the MacLogic environment on Apple Macintosh computers [3]. An older version runs on the ALPES-Prolog environment, on X-Windows workstations [2].

[2] We dub “HyperTracing” the combination of traditional execution tracing, across different program executions through common database side-effects, browsing of the effects of I/O predicates, program source browsing, and navigation with the help of declarative debugging algorithms.

6.1. Theory Summary

Here's our **declarative execution debugging** theory in a nutshell. As we move to the right on the table, cells denote *additions or changes* to those to their left.

Language:	**Normal programs**	**Partial relations**	**Cuts**	**Output side-effects**	**Database side-effects**
Goal behavior facets that the user sees	Solution, solution set; variables in terms are universal logical variables	Call	Terms are just patterns: variables are not logical	ibidem, plus output segment	
Bug instances	Wrong clause instance; incomplete predicate instance	Inadmissible subgoal instance		Bad output predicate instance	
Rules in the oracle theory	User statements are oracle statements; oracle consistency; solution sets stated incomplete contain only solutions stated correct; generic subsumption rules; NAF rules; subsumption among solutions; goal with a solution identical to it can't have an incomplete solution set	The correctness of solutions and solution sets of inadmissible goals is undefined; subsumption among goal calls	A solution or call "subsumes" another only if it is identical (modulo variable renaming)	Identical output "subsumption" rule	
Suspect tree node types	Solution, solution set	Call		Output segment	Additional call nodes for relevant side-effect goals

Table 6.1: Theory summary for declarative execution debugging

For all language features discriminated in the columns above, *suspect sets, bug instance search trees and algorithms are identical*, relatively to each suspect tree. The main algorithms for declarative execution debugging are:

• AD&Q: Abstract Divide&Query, essencially Shapiro's Divide & Query abstracted from suspect type, and also ignoring tree form. Minimizes the number of suspects after a single query in the worst case, not the number of queries.

• GD&Q(N): Generalized Divide&Query with N-ply lookahead, using tree form factors. Minimizes the number of queries necessary for diagnosis, in the worst case.

• PD&Q(N): Probabilistic Divide & Query with N-ply lookahead, using information about the probability of suspect tree nodes being correct/incorrect. Minimizes the expected number of queries necessary for diagnosis, in average.

For **Declarative Source Debugging**, the table above is valid except for the "bug instances" row, substituted now by a "bugs" row (because we search for bugs rather than bug instances):

Language:	**Normal programs**	**Partial Relations**	**Cuts**	**Output side-effects**	**Database side-effects**
Bugs	Wrong clause, Incomplete predicate	Inadmissible subgoal		Bad output predicate	

Table 6.2: Theory changes for declarative source debugging

Source suspect sets, bug search trees and algorithms are also identical for all languages, relatively to each suspect tree. The main algorithms for declarative source debugging are:

• SD&Q(N): Source Divide&Query with N-ply lookahead, using tree form factors defined over collapsed suspect trees (suspect trees where adjacent nodes, that are instances of the same program component, are collapsed into a single node). Minimizes the number of queries necessary for a (source) diagnosis, in the worst case.

• PSD&Q(N): Probabilistic Source Divide & Query with N-ply lookahead, using information about the probability of suspect tree nodes being correct/incorrect. Minimizes the expected number of queries necessary for diagnosis, in average.

• N&Q: Narrow & Query. Minimizes the number of (source) suspects left after a single query.

• SECURE(N): SECURE algorithm, eventually reverting to GD&Q(N) in the final phase. Similar to N&Q initially, but assuming that a program component with a computed instance under a correct goal behavior facet is correct. If the assumption is found invalid, reverts to GD&Q(N).

6.2. Design goals

This was our main goal during the implementation work:

• To exemplify the present debugging approach, implementing most of the theoretic framework in a modular prototype written in Prolog, offering an easy to use graphical interface.

• To adopt an implementation design allowing in the future debugging of large computations (cf. chapter 7).

We consider the user interface issue very important. Previous declarative debuggers made the user just a passive oracle:

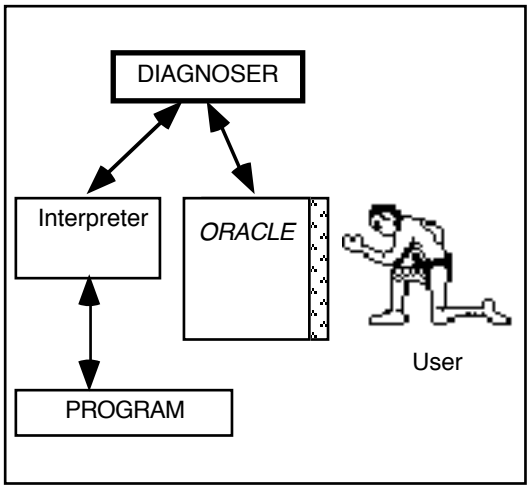

Figure 6.1: Old interface paradigm

This corresponds to a rigid control cycle:

• select a goal behavior facet using the diagnosis algorithm

• query the user

• continue if not a diagnosis

We aimed at giving the user more control, allowing him to shift between his oracle role, with the debugger working as an "automatic pilot" navigating the execution, to a more conventional role where the user is the pilot himself, more like in non-declarative debuggers:

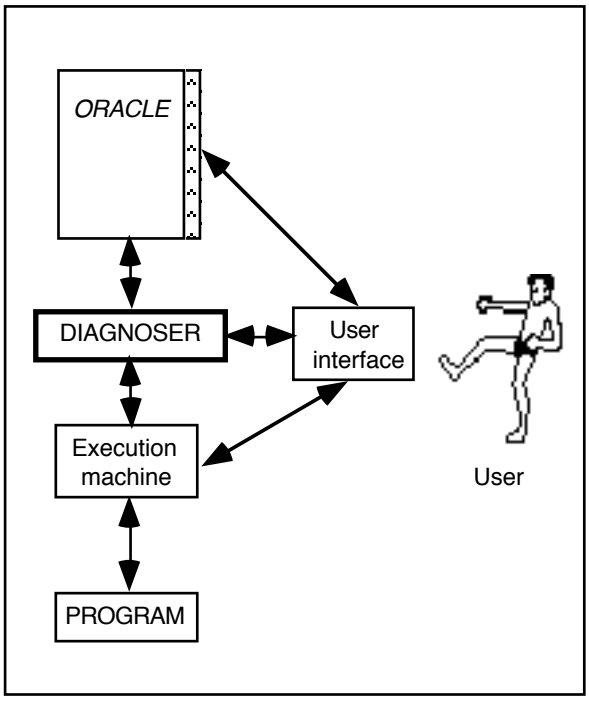

Figure 6.2: New interface paradigm

Rather than being queried in sequence about goal behavior facets, the user now (optionally) states declarative information about them, and may eventually require the debugger to show him another "interesting" goal facet. The debugger will show it to him, or a diagnosis if possible, but will *not* force the user to continue answering queries. In other words, the rigid control cycle above is broken into separate pieces.

The debugger now must have the ability to follow, simultaneously, different "debugging threads", in the sense that the user can give statements as oracle, or ask to examine a particular region of the execution (i.e., using a non-declarative command), or ask the debugger to exhibit an interesting node (by applying a single iteration of any of the declarative debugging algorithms[1]) - all this in arbitrary order, and potentially for different top goals.

[1]The user may even use *different* debugging algorithms while debugging the same top goal. Their common ground is the evolving oracle theory, which is used in all algorithms.

This looser style of interaction combines well with a direct manipulation graphical user interface, where goal behavior facets, oracle statements and other concepts can be represented by an explicit graphical object. Most user commands can therefore be given using menus, applied to a selected object.

6.3. Functionality, vis-à-vis the theory

Here's an overview of the HyperTracer, regarding its declarative debugging capabilities (cf. theory summary tables above).

Language features supported: definite programs, partial relations[1], cuts, output and internal database side-effects; negative literals are not explicitly supported, since nearly all Prolog systems implement NAF with cut. Definite Clause Grammars.

Goal behavior facet types: goal calls, solutions, failure information (i.e., "solution set") and output segment.

Oracle theory rules: oracle consistency; type specifications for some system predicates; "subsumption" among identical goal behavior facets (but not full subsumption; and furthermore, it is being used only to avoid redundant queries, not to discover new bug manifestations - each oracle statement originates a single refined suspect set, which simplifies the implementation).

Declarative Execution Debugging algorithms: AD&Q, or GD&Q(1) without tree form factors[2]; a top-down algorithm[3]. Support for diagnosis explanation and oracle statement retracting.

Declarative Source Debugging algorithms: N&Q and SECURE.

In addition to these declarative debugging features, the HyperTracer has various execution browsing primitives, described below.

---

[1] The HyperTracer currently supports a small set of system built-ins in user programs; this set can be easily enlarged simply by adding entries (clauses) to an HyperTracer global table.

[2] Therefore similar to Shapiro's Divide and Query algorithm ([83], page 44), but for *all* bug types.

[3] A generalized version of the basic top-down algorithm of [4], here applied to any suspect tree.

6.4. Architecture

Following is the global architecture of the HyperTracer debugger:

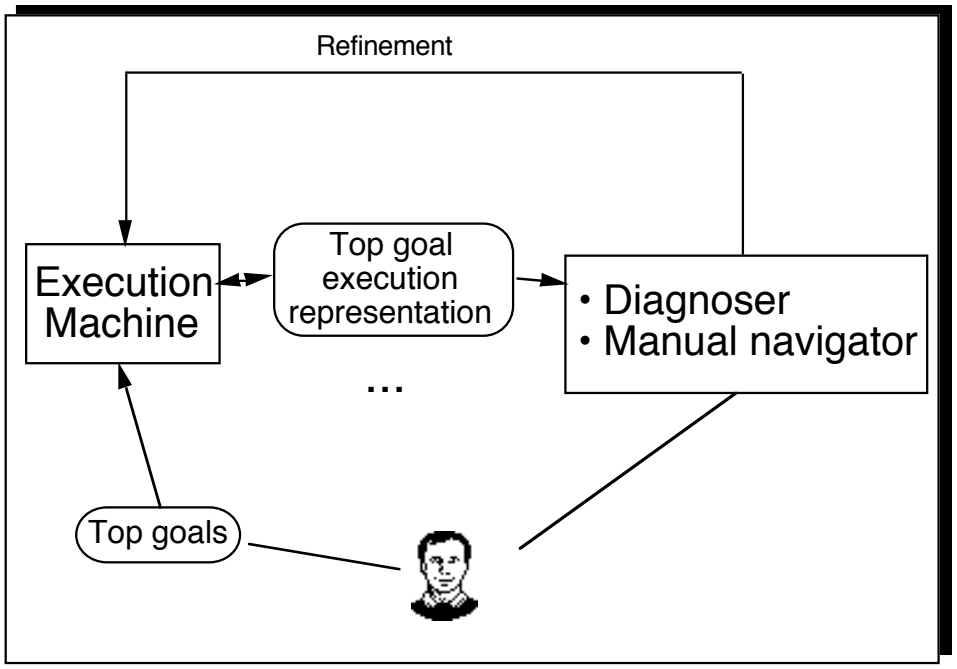

Figure 6.3: HyperTracer architecture

User goals are executed by the "execution machine", a combination of preprocessor + runtime debugger predicates, written in Prolog, that stores *partial* information about the execution in the Execution Database - a set of Prolog relations. The diagnosis and inspection mechanisms use this information, ignoring the details of execution control.

In order to save storage space, only a partial execution representation is kept. Whenever addidional information is necessary, say after the debugger has discarded a bunch of suspects, the Execution Machine is asked to recompute some particular subtree and store additional goal nodes, thus refining the execution database information (cf. next chapter). Side-effect goals are not repeated during recomputations, nor are subcomputations with stored roots, since their results are stored lemmas.

The user interface is based on an object-oriented subsystem also implemented in Prolog, the HyperInterface (described in the "Implementation Issues" chapter). Debugger entities, such as goal behavior, segment, oracle statement, etc. correspond to HyperInterface class definitions and are visualized as HyperInterface objects. User commands are treated as messages to the methods in the class definitions.

6.5. Example debugging session

We'll take a tour with the HyperTracer. Consider the following (buggy) definite clause grammar preprocessor[1]:

```
transform( (A-->B), (NA :- NB), L1,Ln) :- !,
    transform(A,NA,L1,Ln), transform(B,NB,L1,Ln).
transform( (A,B), (NA,NB), L1,Ln) :- !,
    transform(A,NA,L1,L2), transform(B,NB,L2,Ln).
transform( [Terminal] ,true, [Terminal|L], L ) :- !.
transform( A, NA, L1, Ln ) :-
    A=..[Functor|Args], my_append(Args,[L1,Ln],NArgs),
NA=..[Functor|NArgs].


my_append([],L,L) :- !.
my_append([X|L1],L2,[x|L]) :- my_append(L1,L2,L).   %Bug: x instead of X

load_grammar([Rule|Rules]) :- !,
    transform(Rule,Clause,_,_), assert_user(Clause), load_grammar(Rules).

load_grammar([]).

a_phrase(G,L,Remaining) :- transform(G,NG,L,Remaining), NG.
```

`assert_user` is the HyperTracer-supportted assert predicate.

After loading this program with the HyperTracer (so that it becomes preprocessed and ready for debugging), we load a grammar with the following top goal, executed under the HyperTracer:

```
?- tg( % HyperTracer command to execute a top goal
load_grammar([
(s(sentence(N,V)) --> np(N),vp(V)),

(np(noun_phrase(D,N,R)) --> det(D), n(N), optrel(R)),
(np(noun_phrase(P)) --> pn(P)),

(vp(verb_phrase(V,N)) --> tv(V),np(N)),
(vp(verb_phrase(V)) --> iv(V)),

(optrel(relative)),
```

---

[1] The HyperTracer has its own (correct!) DCG preprocessor, plus an oracle pretty-printing facility (cf. “Intensional oracle statements”, chapter 3). Here we're using an incorrect preprocessor, running as an user program, to show the HyperTracer dealing with database side-effects.

```
(optrel(relative(V)) --> [that], vp(V)),

(pn(mc) --> [miguel]),
(pn(lmp) --> [luis]),

(iv(debug) --> [debugs]),

(tv(use) --> [uses]),
(tv(debug) --> [debugs]),
(tv(contain) --> [contain]),

(det(a) --> [a]),
(det(a) --> [some]),
(det(all) --> [all]),

(n(debugger) --> [debugger]),
(n(program) --> [programs]),
(n(bug) --> [bugs])
]) ).
```

And now that the grammar is loaded, let's try to parse a phrase, by executing another top goal:

```
?- tg( a_phrase(s(Semantics),[miguel,uses,a,debugger,that,debugs],[]) ).
```

The parsing succeeds, but the first solution is incorrect. The Semantics argument is returning a free variable, whereas it should return a parse tree:

```
Semantics = _106
yes
```

Enter the HyperTracer. Here's its window with the top goals executed under the HyperTracer (in abridged form, with '…' in place of very large terms):

Figure 6.4: Top Goals window

By double-clicking on the a_phrase goal to inspect its behavior, we get its goal behavior window[1]:

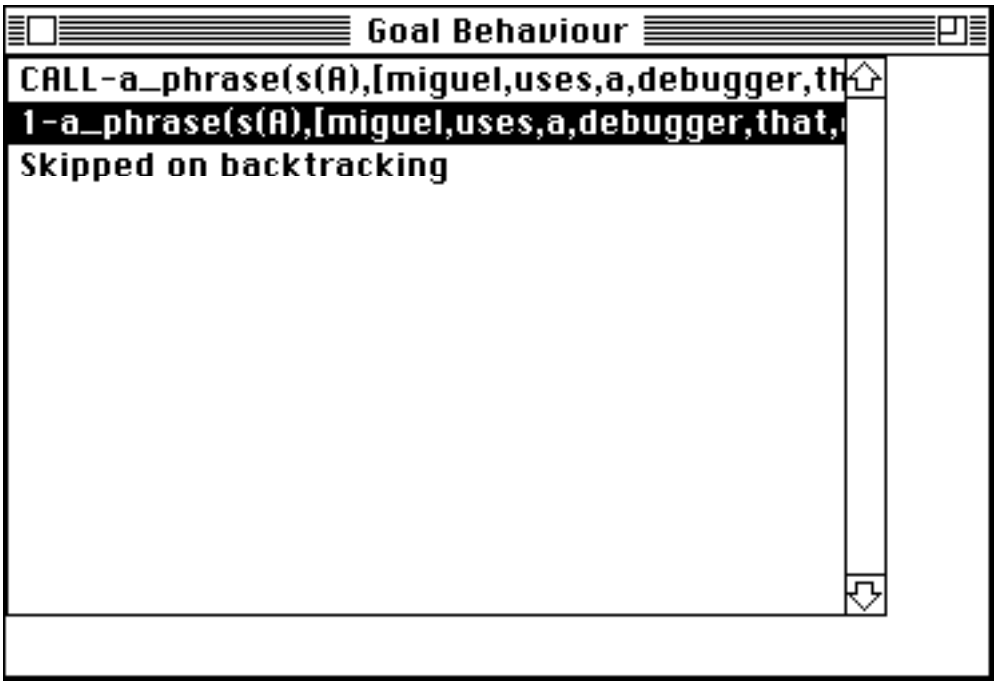

Figure 6.5: A Goal Behavior window

Three goal behavior facets are shown: the goal **CALL**, the **1**st solution, and information about the solution set, or in this case, lack of it: since we didn't ask more solutions to the Prolog top level shell, after we detected the first to be wrong, the goal had no more chances to produce more, and hence was **skipped on backtracking** by the Prolog top level shell: no point in complaining about missing solutions!

Let's try to find a reason for the incorrect solution. We'll use the Narrow&Query declarative source debugging algorithm, using the SECURE assumption:

```
?- use_SECURE_assumption.
yes
```

[1] Goal behavior facets that are partially obscured can be made visible by demand; the cosmetics of the HyperTracer interface needs some improvements, which are naturally independent of the debugger functionality… cf. “HyperInterface” section, “Implementation Issues” chapter.

After having selected the solution facet in the previous window, we ask WHY the solution was produced by using the “WHY, N&Q” command in a menu (meaning, “use the N&Q algorithm to diagnose the cause for the incorrect solution”) and the HyperTracer selects the next goal solution facet:

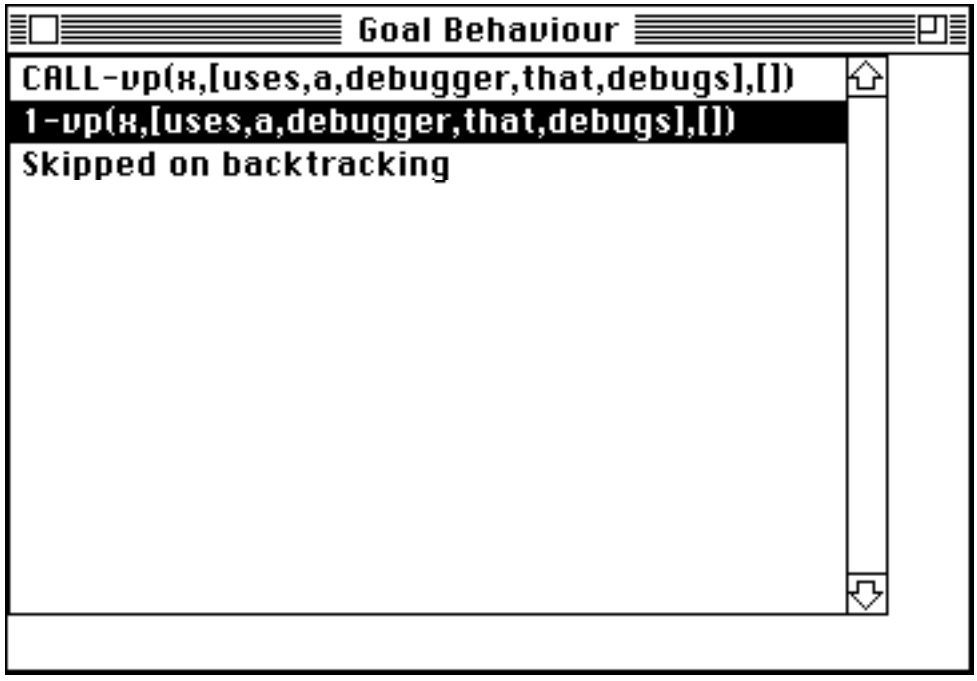

Figure 6.6: Showing a solution

Again, the selected solution is incorrect ('x' is not a parsing tree), so we continue with “WHY, N&Q”:

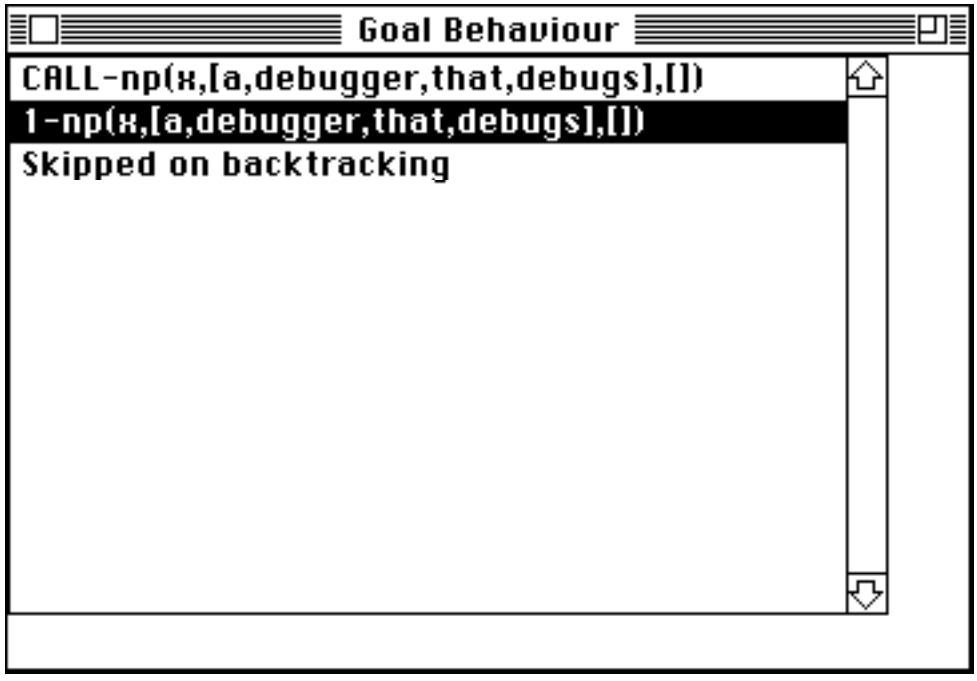

Figure 6.7: Showing a solution

And again:

Figure 6.8: Showing a solution

The Hypertracer now has isolated what might be considered a diagnosis. But it knows better:

Figure 6.9: Debugging across side-effects

After clicking OK above:

Figure 6.10: Showing a goal call

The HyperTracer *has taken us automatically to a suspect goal call in the first top goal execution* (the grammar loading). The call is inadmissible, because optrel should have identical 2nd and 3rd arguments, and the semantics argument should not be 'x'. After choosing “WHY, N&Q” again:

Figure 6.11: Showing a solution

The transformation performed by the preprocessor is incorrect, and we continue:

Figure 6.12: Showing a solution

This append solution is correct. After “Correct & Continue”:

Figure 6.13: Showing a solution

But this one isn't: the third list should contain 'relative' instaed of 'x'. After “WHY, N&Q” once more, we get the diagnosis in the source program window:

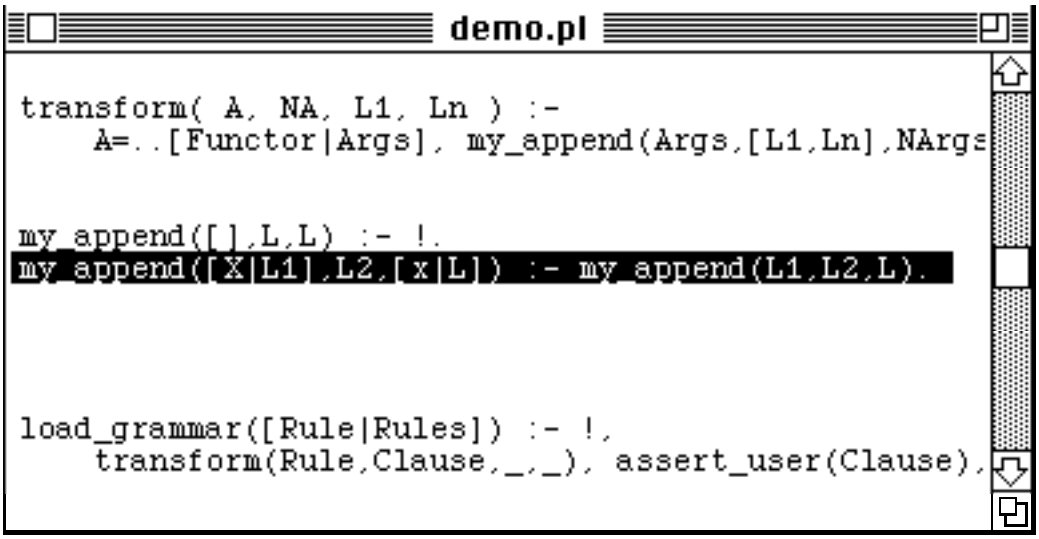

Figure 6.14: Showing a wrong clause

Eight queries were necessary (counting the "top goal incorrect" statement) to find a diagnosis, using declarative source debugging. The SECURE assumption was valid in this case, and no confirmation was needed. AD&Q, the main declarative execution debugging algorithm available in the HyperTracer, would need 11 queries.

6.6. User interface object types

In this subsection we review some of the more relevant HyperTracer interface object classes, i.e. debugger interface objects which the user can act on, namely those with a strong meaning in terms of Prolog debugging. The interface browsing features are implicit in the HyperInterface and MacLogic, and are not described here.

You may look into appendix C for an exhaustive list of object classes and available commands.

HyperTracer control window

This is a window giving access (by doubleclicking a list item) to the 3 main HyperTracer windows.

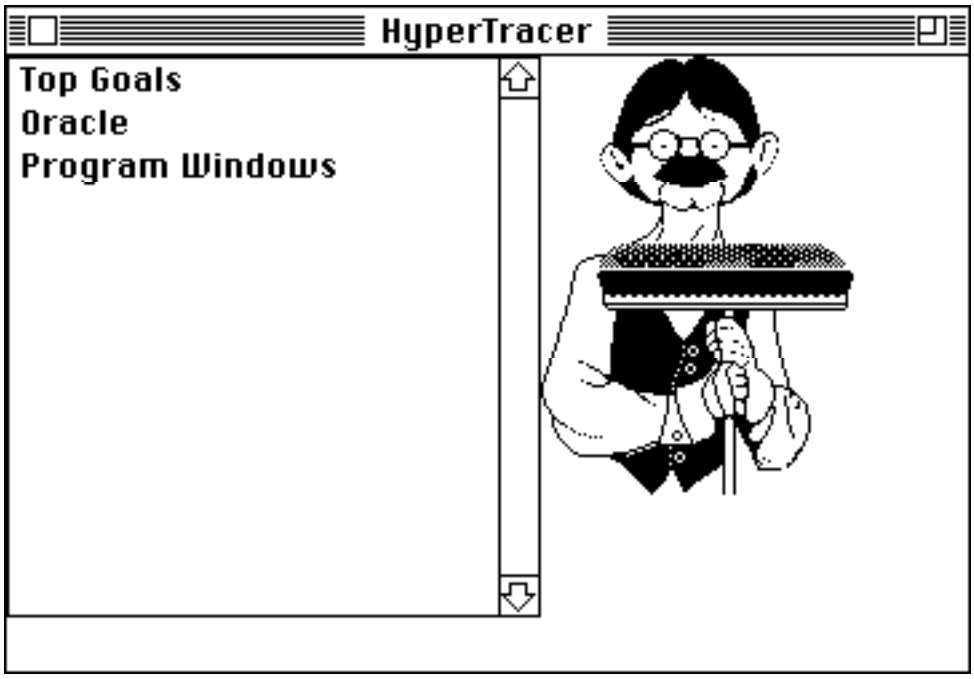

Figure 6.15: The HyperTracer control window

"Program Windows" window

A window with a list of all windows containing program files to be debugged. A doubleclick on a file opens its source window.

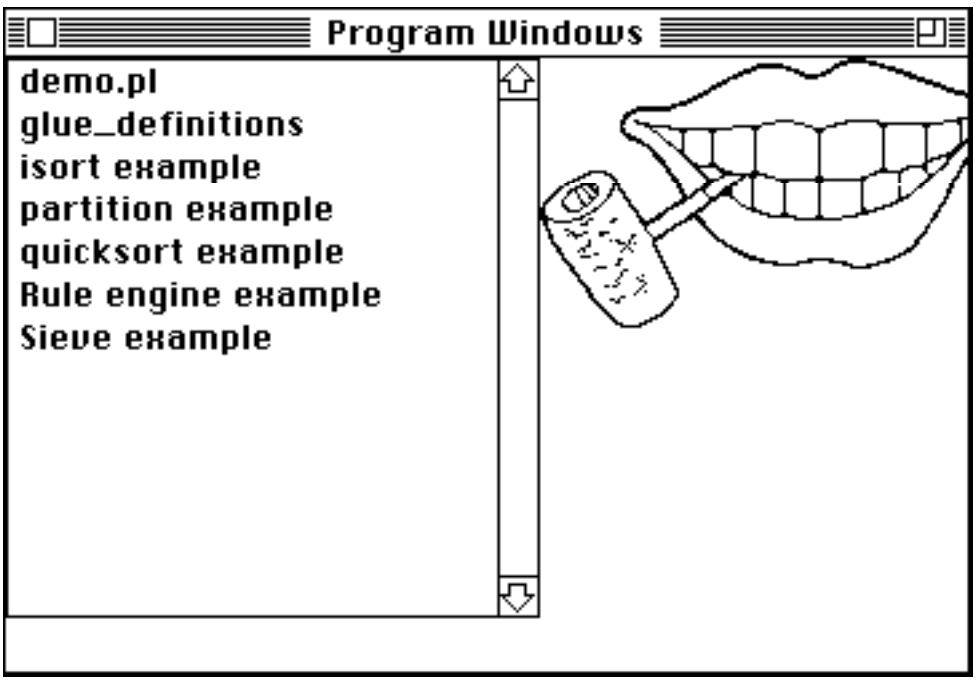

Figure 6.16: The "Programs Windows" window

Window with list of top goals

Lists all top goals executed under the HyperTracer (cf. figure at the beginning of the example session). Double-clicking on one of them opens or selects its goal behavior window (cf. below).

Oracle window

Contains an abridged summary of all oracle statements. A doubleclick on a statement opens the goal behavior window where the selected oracle statement was made, and selects the appropriate goal behavior facet. There's also a “Remove Statement” command, which retracts the selected oracle statement (for when the user makes mistakes as an oracle). Here's the oracle window after some of the user interactions reported above :

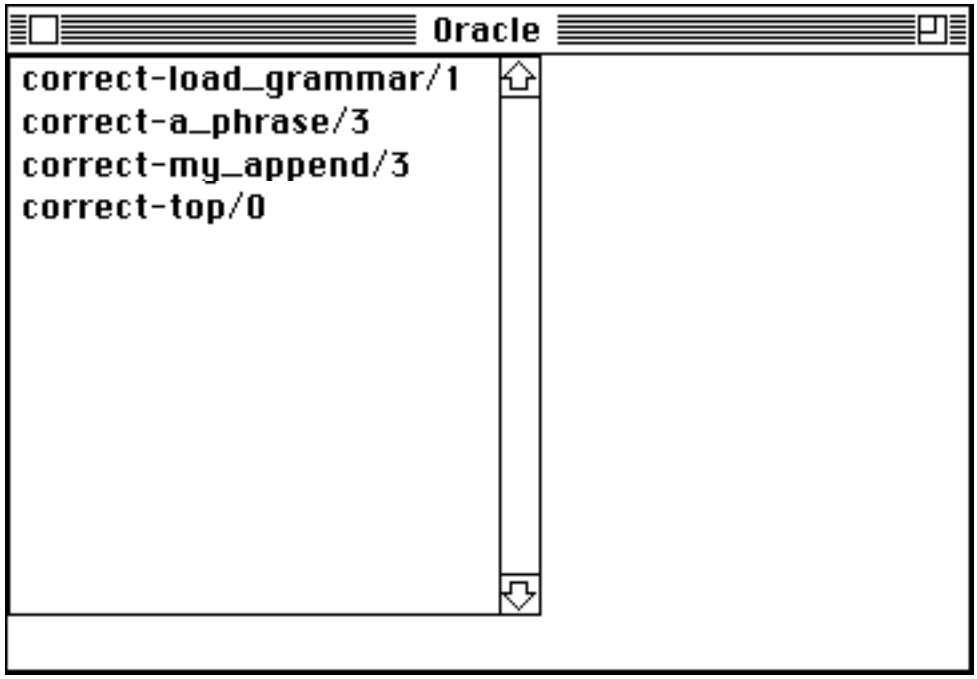

Figure 6.17: Oracle window

Notice that there are no incorrect statements: during the session above we just used the WHY command, which only *assumes* a goal behavior facet to be incorrect.

Goal behavior window

Contains all declarative information about a goal: its call, the solutions (numbered) and failure information, all as individual list items.

Some facets may be missing (for example, there are all solutions but the first, and no call). To avoid massive storage of everything, the HyperTracer typically stores only some goal facets at first. One can request recomputation of the missing ones for a particular goal, by using the 'Complete Information' command (or the declarative diagnosis will do this automatically when necessary).

The menu commands that apply to the whole window, or specifically to the above goal behavior facets, allow additional information to be accessed, such as side-effects, source, and -last but not the least- the bulk of the navigational and diagnosis features of the debugger.

Window with DB side-effects

Contains all database side-effect calls changing a predicate. Only those older than a certain time - i.e., the relevant ones - are displayed.The same commands that apply to goal behavior facets (in goal behavior windows) are available for these goal calls. Here are the side-effects potentially affecting the `optrel` solution in the HyperTracer example session above:

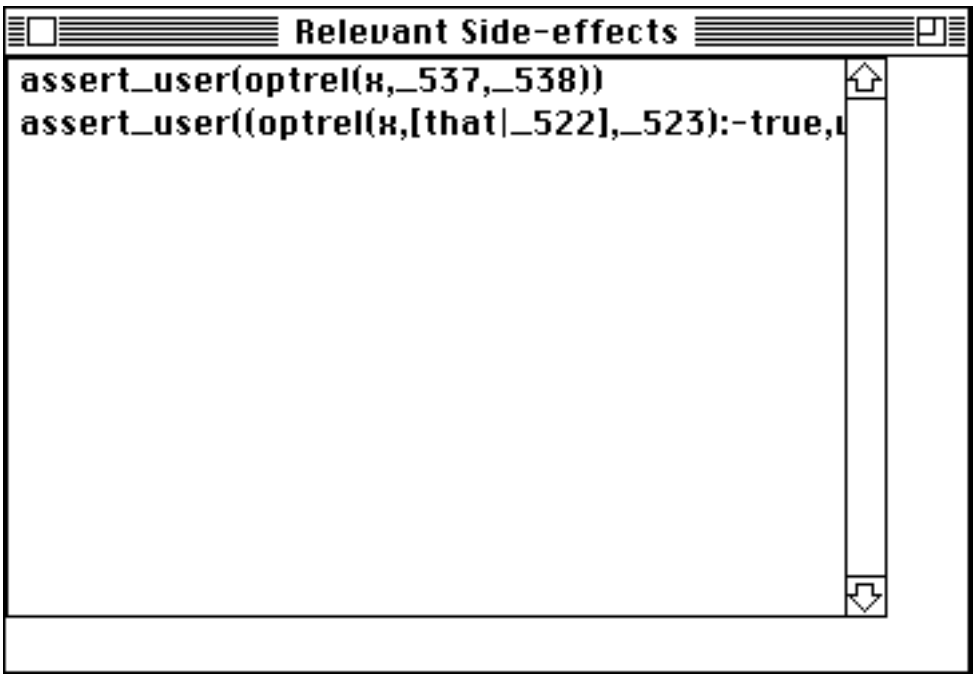

Figure 6.18: A DB side-effects window

Window with output side-effects

Contains all output side-effect calls under a goal (its segment), or under it and up to the computation of one of its solutions.The same commands that apply to goal behavior facets (in goal behavior windows) are available for these goal calls.

For example, consider goal top for the following program:

```
p :- write_user(1).        p :- write_user(2).
q :- write_user(3).              q :- write_user(4).
top :- p,q,fail.
```

Here's its output side-effects window:

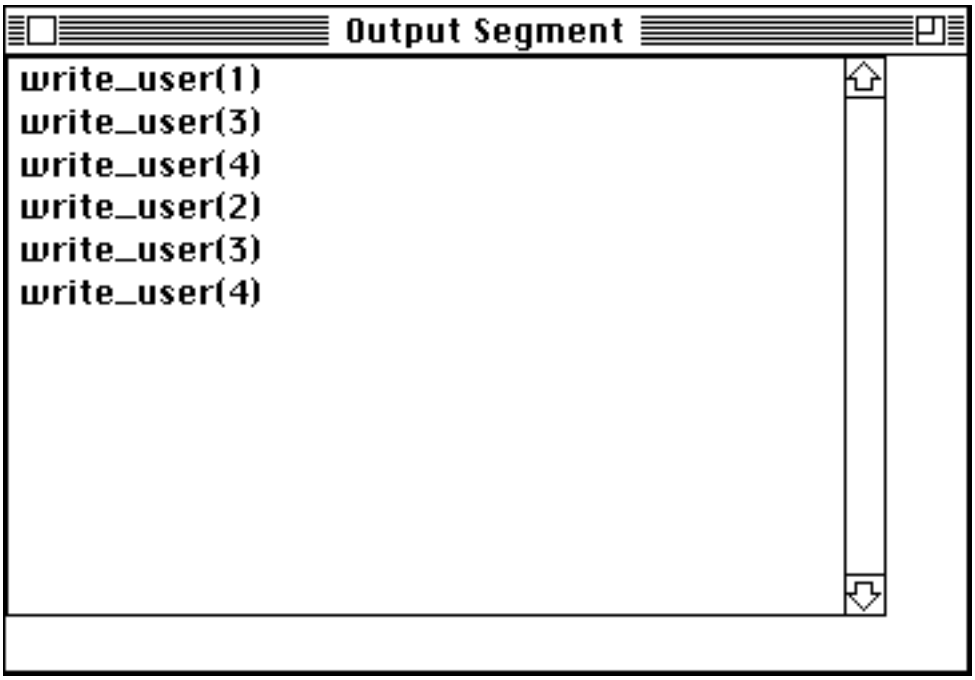

Figure 6.19: An output side-effects window

Window with undefined predicates

A window with all predicates undefined in the currently loaded program, obtained by using a menu coomand in the “Program Windows” window. A doubleclick on a predicate opens the source window containing a clause calling it, and selects it.

“WHY filter” window

It allows you to specify and impose a Prolog term which must occur in all suspects (except output segments) to be considered by the WHY commands, by editing the text field and pressing “This filter”. Whether the filtering by WHY commands is enabled or disabled depends on whether the “Apply” checkbox is checked or not. Following is its default state: disabled, and furthermore with a variable (“Any”) which would match any term.

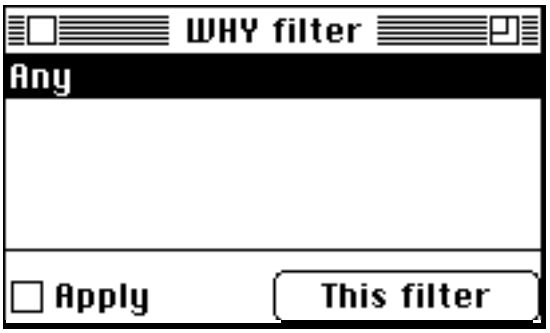

Figure 6.20: The "WHY filter" window

Window with a tree view of a Prolog term

A window representing a Prolog term in tree form. Can be created for any (goal behavior facet) term.

6.7. HyperTracer commands

All but the first two are available from pull-down menus, applied to a selected goal behavior facet.

6.7.1.Main execution browsing commands

tg(G) (a Prolog meta-predicate)

Executes goal G under the debugger, requiring the interesting parts of the program to have been loaded by the debugger. G is implicitly stated admissible. Any number of solutions can be computed, and execution can be interrupted by the user with the keyboard (or by user specified spypoints conditions, or type violations - cf. below). The execution database is always kept in a consistent state.

Declarations

There are essencially 3 types of textual declarations available to users:

• Directives to make a goal immediately accessible for debugging (i.e., turning it into an access point, cf. "Virtual Trace Storage" chapter 7). For example, the HyperTracer already has declarations like the following for all side-effects, to avoid their recomputation:

```
visit_solution(asserta_user(_)).
```

• Conditional spypoints can also be specified, as a particular case of the previous declarations. The HyperTracer contains some (type-checking) declarations as follows:

```
visit_call(assert_user(X)) :- var(X), !, stop_at_once.
```

• Correctness declarations for the HyperTracer diagnosers, using the `correct_solution( Goal, Solution_number )` predicate. For example, all solutions of `asserta` goals are correct:

```
correct_solution(asserta_user(_),_).
```

AND tree

Opens a window with the and proof tree of the selected solution (coinciding with the suspect tree ignoring the effects of cuts). A double_click on a tree node opens the goal behavior window containing the goal facet represented in the node, and selects it.

Latest solution below

An example of a (back)tracing command. Selects the last goal solution under the selected one. If the respective goal behavior window does not exist it is created.

Complete information

Completes the information in a goal behavior window, by using goal recomputation to “fill in” the missing pieces of information.

6.7.2.Main declarative debugging commands

These commands are mostly available for goal behavior windows.

Correct

Declare the selected suspect (goal behavior facet) as correct. This is a way to make an oracle statement. After checking for oracle consistency, an abridged reference to the statement is made in the Oracle window.

Why, D&Q

Builds the suspect tree rooted in the selected goal behavior facet (suspect), refines it using the current oracle knowledge, eventually refines it further by using the (cf.) "WHY filter" and chooses a suspect according to a particular diagnosis algorithm. Then it opens a goal behavior window with the chosen suspect selected. If there's only one suspect (a diagnosis), the relevant program source is shown.

This variant uses the Abstract Divide & Query algorithm (GD&Q(1) ignoring tree form), and considers the "pure" version of the current suspect set (ignoring the effect of Prolog cuts, but not of database side-effects). There's a different command (WHY, D&Q…) considering the effect of cuts.

Why, N&Q

A variant of WHY, the main command to be used in debugging. "Why, N&Q" and "Why…, N&Q" are the *declarative source debugging* commands available in the HyperTracer. They require the execution database to be completly explicit - no recomputations to be needed (cf. HyperTracer architecture description above).

It builds the suspect tree rooted in the selected goal behavior facet (suspect), refines it using the current oracle knowledge, eventually refines it further by using the (cf.) "WHY filter", obtains the source suspect set, and chooses a suspect according to the Narrow&Query algorithm. It will or not use the SECURE assumption (thus behaving like the SECURE(1) algorithm), depending on the previous use of the "use_SECURE_assumption" command (cf. above). Then it opens a goal behavior window with the chosen suspect selected. If there's only one suspect (a diagnosis), the relevant program source is shown.

This variant considers the pure version of the current suspect set (ignoring the effect of Prolog cuts).

Correct & continue

This is the other command (in addition to Why) typically used during algorithmic diagnosis.

It is a combination of commands "Correct" and WHY, to make an iteration in an ongoing algorithmic diagnosis process (no matter the algorithm being used). It is equivalent to the following steps:

1) Apply the (cf.above) Correct command to the selected suspect (goal behavior facet), and set its window aside.

2) Find the suspect which was "insinuated incorrect"[1] (with a command WHY) and which caused inspection of the present one via some diagnosis algorithm.

3) Apply the *same* WHY command to that suspect, causing a different suspect (or even a diagnosis) to be inspected, due to the new oracle statement and consequent suspect set refinement.

This command cannot be applied to output segments.

Correct segment

Declare the selected suspect (on the side-effects in the window) as having a correct output segment. An abridged reference to the statement is made in the Oracle window.

Why this segment

A variant of WHY (cf. command "Why, D&Q" above), now applied to the goal segment.

This variant uses the GD&Q(1) algorithm.

Explain diagnosis

This command is inspired on the discussion in section "Inconsistent oracle", chapter 2. It gives an explanation for the current suspect set for the selected goal behavior facet, ignoring the suspects originated by extralogical features. There's a variant of this command for less pure suspect sets, taking Prolog cuts into account: "Explain diagnosis…". The current suspect set may have just one element, being a diagnosis - hence the command name.

---

[1] The WHY command may be used even without the corresponding oracle statement, hence "insinuating" the selected goal behavior facet to be incorrect.

The explanation is the minimum[1] set of oracle statements necessary to obtain it. It is displayed by setting aside (nearly) all HyperTracer interface objects, except the goal behavior windows where the relevant oracle statements were made. This allows the user to retract an (oracle) statement causing an unpleasant diagnosis. In other words, one made a mistake as oracle and is given a chance to retract himself - cf. "Oracle window" above.

This command does not currently support declarative source debugging algorithms.

6.7.3. Commands for browsing across program changes

In addition to implicit browsing across program changes, available when using algorithmic debugging of programs with database side-effects, the user can perform browsing explicitly:

DB SEs on me

Opens a window containing a list of all DB side-effect calls potentially affecting the predicate of the selected goal behavior facet (cf. "Window with DB side-effects" above). Only those calls made previously to the facet time stamp are shown, since no others should be relevant for that facet.

6.7.4. Commands for browsing output side-effects

Output SEs under me

Opens a window containing a list of all side-effects under the selected goal behavior facet (cf. "Window with output side-effects" above). If the selected facet is the call or the failure, all are shown. If it is a solution, only those that occurred (chronologically) up to it are shown.

6.7.5. Source browsing commands

Matching clause

Opens the source window containing the clause matching the selected solution facet, and selects it.

---

[1] Assuming that no subsumption rules are being used by the oracle, which simplifies the implementation: the relevant oracle statements are simply those concerning the literals in the diagnosis.

Matching clause producer

Shows the assert call producing the clause matching the selected solution, or a message if the clause was not asserted dynamically (“You produced it!”).

Parent clause

Opens the source window containing the clause calling the goal in the window, and selects it.

7. Implementation issues

In this chapter we overview the possibilities for implementing access to computation traces and present our chosen method of doing it (first described in [12]). We also overview the HyperTracer design, including its human interface subsystem. We conclude with comments on performance.

7.1. How to access a computation trace ?

The typical straightforward way to examine Prolog derivations is through variants of the traditional 3-line interpreter. These variants add extra arguments to convey desired information, which can be examined at runtime (e.g. in a tracer) or at the end of the computation (e.g. in an explanation facility). Partial evaluation can sometimes be used to "compile away" the interpreter to render it more efficient [50].

Future Prolog implementations may come to provide standard low-level support for such meta-level facilities, producing say internal structures for proof trees [5] or for term binding dependencies [10][54]. There's still much to be explored in this vein, by allowing increased access to richer information furnished by Prolog abstract machines. For example, stack frames already include partial information about proof trees, and the trail includes partial information on term binding dependencies. This information could be supplied at the language level for programmers.

Even so, such improvements will still have to deal with situations where the amount of utilized information is large (in particular comprising information about failed derivations, such as suspect trees). Prolog is space efficient because it keeps only the current AND tree, and it certainly would not be so if it kept failed OR branches as well.

Two extreme implementation possibilities come to mind for accessing execution trees, which are exemplified in the next section:

A) database: Execute a top goal once and for all, numbering all its derivation nodes in the order they are visited, and storing in a database full annotations about each one[1].

[1]This approach is taken in [30], and in [34].

B) recomputation: Execute a top goal once and for all, counting all its derivation nodes in the order they are visited; to access information pertinent to some node later on, simply recompute the top goal under the same execution strategy, restarting the node count, and stopping when the desired node number is found. We rely on the basic fact that sequential Prolog implementations are *deterministic*, in the sense that *the same goal call, in the same program*[1]*, will always come up with an identical derivation tree*.

Option **B** is very frugal on space, since it needs barely none at all! However it penalizes the access time to derivation nodes, which is proportional to the size of the original derivation.

Option **A** suffers from a severe problem: the space needed for the database will be roughly proportional to the size of the derivation. One should bear in mind, furthermore, that typically not all derivation nodes need to be accessed.

Initial work on debugging at UNL began by using a meta-interpreter having the diagnosis algorithms embedded in it [63][64]. These relied only on term dependency information, and were not amenable to a uniform divide-and-conquer-based uniform treatment of all bug types.

Work by the author started by using option **B** above [66]. This allowed the first uniform GD&Q(1) algorithm to be implemented, for wrong solutions and incomplete solution sets, for a language with cuts - the "UNL Prolog" dialect [78].

We then proceeded to successive HyperTracer implementations, using successive refinements of the present debugging framework, via a hybrid *compromise* between options **A** and **B**: *by storing some nodes (like one would lemmas) we can avoid the repetition of the computation below them whenever recomputing is needed*. From the user's point of view, the system's response time, once the original computation takes place, can be made reasonable. On the other hand, the diagnosis algorithms make use of the stored information to accept only some abstraction of the derivation tree, till there is further need to focus on some subtree in greater detail[2], whence the necessary partial recomputation is performed on demand.

---

[1]Assuming for now that there are no lasting database or I/O side-effects from one execution to the next.

[2]As when the method of [47] moves from abstract to concrete diagnosis mode.

This idea was first presented in [76], which used it only for binary trees. Our approach can thus be seen as a generalization of his approach.

Recently we used also option **A**, but in a slightly different context: to support updating of logic programs [71] (cf. chapter 8).

We now illustrate the idea of a hybrid compromise with an example.

7.2. Trace access options

Consider the Prolog program and the unsolvable top goal call b(boom):

b(Y) :- b1(Y). b1(Y) :- b2(Y). b2(Y) :- b3(Y).

b3(Y) :- b3_1, b3_2, b3_3(Y).

b3_1. b3_2. b3_3(foo).

This being a nearly deterministic computation, its AND/OR execution tree is easy to depict:

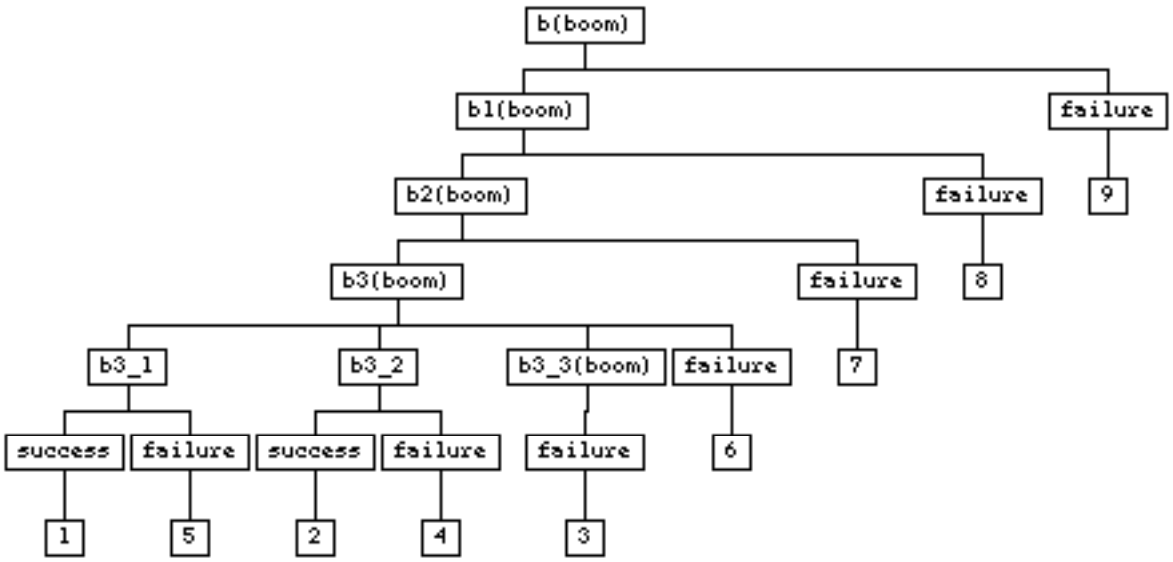

Figure 7.1: An AND/OR execution tree

The numbers show the order in which Prolog visits the tree nodes (EXIT and FAIL ports considered different nodes or suspects). Now suppose we're interested in potentially accessing all solution or failure nodes of this top goal's computation. The **database** option suggests using some mechanism to execute the top goal producing as a side-effect something like:

No.	Type	Information
1	solution	b3_1
2	solution	b3_2
3	failure	b3_3(boom)
4	failure	b3_2
5	failure	b3_1
6	failure	b3(boom)
7	failure	b2(boom)
8	failure	b1(boom)
9	failure	b(boom)

Table 7.1: A simplified execution trace

Assuming we're interested in goal calls only insofar as what regards information impinging on their failure, then solutions and failures alone are (chronologically) counted and kept.

Adopting the **recomputation** option, and if we just retain the goal call `b(boom)`, then we can recover any node in at most 9 steps (counting as steps both solutions and failures).

What we have in mind though is a **hybrid** solution. We keep some nodes in the database, and obtain any others by demand-driven recomputation. In this example we could keep only step 6: any step below it can now be accessed in about half as many steps as before, by considering the intermediate call b3(boom) as a recomputation top goal; on the other hand, if we need to access the steps involving goals b1 and b2 we can use the ("lemma-like") fact that b3(boom) failed without solutions[1], and avoid visiting the steps below it in the above tree.

[1]We can use the step number to identify a lemma. And there's nothing to gain from performing subsumption tests, because Prolog's extralogical features make their use unwarranted.

7.3. Virtual trace storage

The basic idea is to keep an abridged version of the execution tree, stored as a Prolog database only of some selected execution tree nodes (i.e. goal behavior facets). Nodes not expressely stored can be accessed via a demand-driven and shortened recomputation from the stored ones. Changes of criterion regarding which nodes are to be stored is achieved by tuning a tree weight measure bound, and affects the trade-off between the use of database space and expected recomputation time. This parameter, which can be dynamically adjusted, is to be set to an initial value depending on the speed of specific Prolog implementations and their available database space.

7.3.1. Recomputing

In order to recover an intermediate goal call G in the midst of some top goal derivation, it is sufficient to know how many nodes were visited during its complete execution until G (this may include several solutions or solution attempts). Since Prolog undoes all variable bindings on backtracking, logical variables can only be used to number nodes *relatively to the current AND tree*, not all the nodes of an AND/OR tree. The latter numbering can be accomplished with a global counter, to which destructive-assignements are made (i.e. not undone on backtracking).

To reobtain the solutions of a subgoal G' one can of course launch G' directly, instead of the original top goal. However, if one wishes as well to reproduce the goal numbering under G', it is necessary to continue on counting from the numbering performed by its ancestors and previous brothers. This must also be ensured whenever retrying G' for additional solutions.

7.3.2. Choosing access nodes

An **access point** or node is a goal call about which we store some (possibly partial) information - one or more goal behavior facets. Deciding on whether to keep information about a goal G, by turning it into an access point, should take into account the following **constraints**:

a) If we keep G's variable bindings at calling time we can relaunch it later as a goal, visiting an identical computation tree, and therefore gain access to whatever is below G.

b) If we keep the variable bindings of one of its solutions, G', we can omit the computation below that solution whenever we recompute G, or an ancestor of G, simply by making G' a stored access point.

c) If we keep a solution of G, G', but there are subsequent ones to it that were not kept, then, if ever we later recompute G we'll not be able to profit from having stored G'; Prolog's backtracking scheme isn't able to skip the computation of an intermediate known solution within a sequence of solutions simply by storing it as a lemma.

In addition, we wish to **keep the cost of any recomputation under control**. Assuming cost to be measured as the *number of nodes visited*, we arbitrate a maximum number of nodes, the **weight bound** W_{MAX}, as an adopted cut-off measure on the size of recomputations. Furthermore, access point nodes all have just weight 1 within the recomputations that contain them. The cost of recomputing a solution G' for a call G will be the number of nodes $W_{G'}$ visited under G till G' is produced. On the other hand, the cost of rediscovering a goal call B under a call A is the number of nodes W_{AB} visited under A till reaching B.

In order to decide whether to keep a solution G', it is necessary to retain its node count $W_{G'}$ to compare it with W_{MAX}; if $W_{G'} \geq W_{MAX}$, the solution is stored. It is also stored if some previous solution for G was already stored. Similarly for any goal call B under an already stored call A: iff $W_{AB} \geq W_{MAX}$, B is stored.

7.3.3. Time vs space trade-off

The **maximum recomputation cost** for a solution or call is W_{MAX}. As for the **number of access points** created, it depends on the weight bound and its relation to the shape of the execution tree.

Let's assume a *deterministic* (each goal matches a single clause) and balanced execution tree with N nodes. Consider the **number A_S of solutions kept** at access points. **If** $W_{MAX} \leq b$ (the average tree branching factor), **then** $A_S = N/b$: only solutions at the bottom level of the tree will not be kept access points, as the solutions at all other (higher) levels will have weight $\geq W_{MAX}$. **If** $W_{MAX} >> b$ **then** $A_S \approx N/W_{MAX}$, as the solutions kept will be evenly distributed across the tree. This should be the situation for fast Prolog implementations, where W_{MAX} can be made larger without degrading the response time.

Nondeterministic trees degrade A_S, because of the c) constraint above. It forces additional solutions to be stored, no matter how cheap their computation[1].

Now consider the **number of calls kept** in access points, A_C. If the tree consists in just a single chain of nodes, $A_C \approx N/W_{MAX}$. Otherwise A_C will tend to be larger because weights of sibling calls increase to the right: if any one of them is heavy enough to become an access point then all its right siblings will be too[2].

Arbitrating a small weight bound leads to the creation of many access points, hence to greater use of space and smaller recomputation times. A greater weight allows to save memory at the expense of slower performance. Extreme granularities (weight bounds of 0 and ∞) reduce the framework to the "database" or "recomputation" options above.

7.3.4. Side-effects

Side-effects pose additional problems regarding recomputation: we wish to inspect Prolog's wanderings without fiddling with the world state, which may have changed during the original execution through side-effects. When side-effects are external (**I/O**), it is sufficient to enforce their derivation nodes to become appropriate "lemma" (or "access point") nodes. Proceeding this way we can access and recompute their ancestors, and simply look up the previously logged side-effect goal solutions, thus avoiding duplicate side-effect interactions with the external world. In other words, during recomputations output is inhibited and input is consumed from logged information.

---

[1]Extending Prolog to allow a sort of "goto next redo", relatively to a stored solution, would solve this problem by allowing skipping of adjacent ("ORwise") solutions during recomputations. This possibility would imply costs in state-saving however; we know of no Prolog implementation providing such a facility.

[2]It would be nice if applications could make do without storage of calls… For example, debugging of normal programs (typeless, and without cuts or side-effects) can do without it. Another possibility to avoid storing calls would be to undo the bindings done by the computation under a stored solution: if the stored solution includes term binding dependencies, one can recover the original goal call.

Internal (**DB**) side-effects complicate matters. Assuming we have a single top goal (i.e. alternatively we reinstate the whole database when moving to the next top goal), and assuming the program doesn't tamper with executable clauses but merely with the so called internal database, we could then treat DB side-effects in the same way we do I/O ones. Either assumption is too restrictive however.

As we're interested in inspecting several top goals for the same program, “internal” side-effects cannot simply become access nodes. Some indexing relative to the current top goal is needed, or else a global goal numbering across top goals. On the other hand, self-modifying code, a possible practice for Prolog systems as a whole, complies us to make clauses valid only for the “time interval” between 2 goal numbers (of the `assert` which creates the clause and of the `retract` which eventually destroys it), these being potentially under different top goals.

We solved both difficulties by *marking clauses, on their creation and destruction*, with the (global) goal number(s) of their creator/destructor goal calls. This way we can constrain clauses to be used only if they “exist” at the time they're potentially matched, and avoid problems with interaction among different top goals (we use in fact a single goal numbering across all top goals). Furthermore, the assert and retract calls are always stored as access points.

7.4. The HyperTracer design

Following is the global architecture of the HyperTracer debugger, repeated here for convenience, as in figure 6.3:

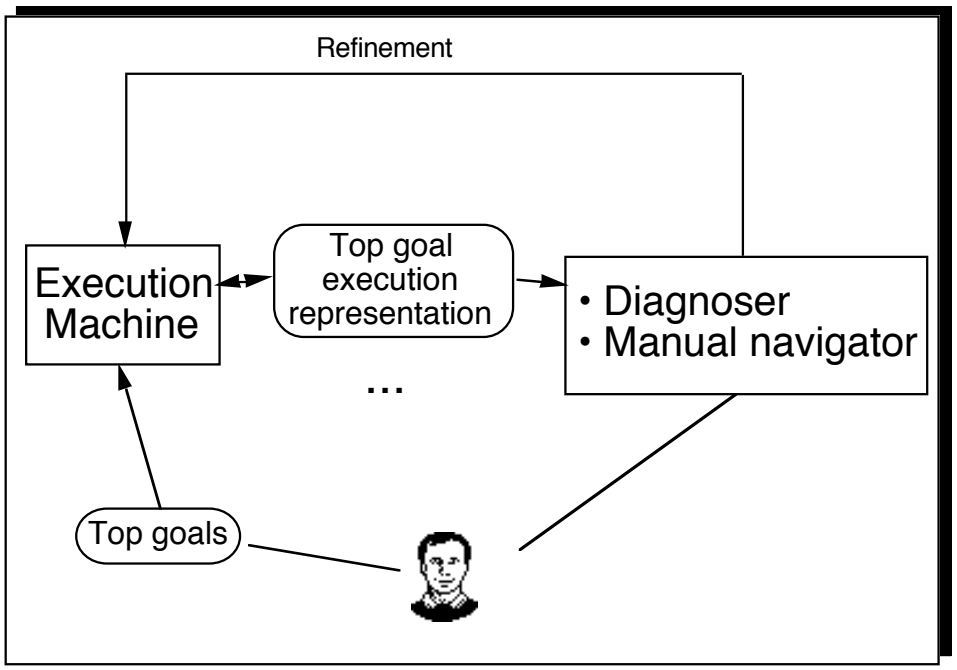

Figure 7.2: HyperTracer architecture

User goals are executed by the "execution machine" (currently a combination of preprocessor + runtime debugger predicates), that stores partial information about the execution in the Execution Database. The diagnosis and inspection mechanisms use this information, ignoring the details of execution control. Whenever addidional suspect nodes are necessary, they can ask the Execution Machine to recompute a particular subtree and store its goal nodes, refining the execution database information.

Side-effect goals are not repeated during recomputations, nor are subcomputations with stored roots, which are used as lemmas.

7.4.1. Execution Database

In addition to some information regarding top goals, the Execution Database contains mainly information grouped into "access points".

7.4.1.1. Access point (ap) overview

An *access point* is the set of all stored information about a goal call. An *ap facet* is the chunk of information relating either to the call, each of the solutions or the failure. It is identified by its *time* moment of execution.

Only current access points are considered as suspect goals, and *only atomic goals can be access points*, as only atomic goals are queried about by the diagnosis algorithms. To suspect sets restricted to access points we call *abstract* suspect sets, as oposed to *concrete*[1] suspect sets (these corresponding to a weight bound of 0 in the execution machine, when all goals are aps).

Following are the Prolog relations used to represent aps. Suspect sets are not explicitly kept in aps, to minimize execution space and time overheads. Abstract suspect sets are easily reconstructed from the relations below, only when needed.

Side-effect goal calls (both internal and I/O) are always made complete access points, and program clauses are taken valid only within the "time interval" in which they exist, as discussed above.

7.4.1.2. Ap identification

An ap is identified by the time of the goal call in the original computation. Each ap facet is represented by its own time (an integer):

```
an_ap(AP)
```

7.4.1.3. Zooming status

This is the information telling when a zoom-in operation (cf. below) would be relevant, i.e.there are still nodes to be made aps in this execution region.

First, a "flag" saying if there are any nonstored solution or failure nodes between this call and the aps (failure/solutions) immediately below it:

```
top_down_concrete(AP)
```

Second, a similar "flag" saying if the nearest call stored (i.e., this ap's "launcher for recomputations") is its father; if not, there are ancestors still candidate for storing:

```
father_is_launcher(AP)
```

7.4.1.4. Call information

Goal call term:

```
call_info(AP,Term)
```

---

[1]Following [47] terminology.

The previous ap facet in the current derivation (i.e., in the same partial AND-tree):

```
previous(AP,Previous)
```

The reference of the father clause:

```
father_clause(AP,FC)
```

If this call is before an "uncle" cut, the time of matching of the clause instance containing that cut:

```
before_cut(AP,Match_time)
```

If the goal is an output side-effect:

```
a_side_effect(Time)
```

7.4.1.5. Solution information

Goal solution term, time of solution EXIT port (LT):

```
solution(AP,LT,N,Term,Matching_clause)
```

Entering time (ET) for the computation fragment producing the solution (only for solutions 2 and later, if any), for later identifying what's under the goal:

```
computation_fragment(AP,ET,LT,N)
```

Again, the previous ap facet in the current derivation, possibly (but not necessarily) under the goal:

```
previous(LT,Previous)
```

7.4.1.6. Failure information

Time of FAIL port (LT), number of goal solutions, reference to the matching predicate definition:

```
failure(AP,LT,N_sols,Functor/Arity)
```

Time of REDO port (to later identify what's under the goal):

```
computation_fragment(AP,ET,LT,failure)
```

Previous ap facet of the goal call (present iff the goal failure facet is stored but the goal call isn't), necessary to know which goal to use later as a launcher of a recomputation to obtain the present call:

```
previous_for_launcher(Call_ap,AP)
```

7.4.1.7. Global and other information

Top goal time interval

```
top_goal_time(Start_time)     top_goal_end(Start,Stop_time)
```

Comments on the “previous” relation

This relation represents the SLDNF OR tree at the current level of abstraction. Nodes in the relation represent either call or solution ap facets. Each of these “has” a single associated tuple, where it is the first argument.

Notice that previous/2 is not transitive, but one can define a transitive “same derivation” relation by using the REDO/EXIT times of solutions. One can also define an “ancestor” relation from previous/2 together with this time information.

7.4.2. Execution Machine

The execution machine is currently implemented in C-Prolog, without specific builtins except for a set of primitives to destructively assign integers to a variable (used mostly to implement non-backtrackable counters).

7.4.2.1. Program representation

The program being debugged is assumed to be preprocessed and loaded (and compiled, when using a compiler-based Prolog implementation), in order to transport debugger information in additional predicate arguments, and to have a few “hook predicates” called at the call/fail, exit/redo and clause match ports of program predicates.

For each predicate definition a new predicate symbol is created, invisible to the user, and the original clauses are changed to have it in their heads. The original predicate symbol is just used for an (also new) “glue” predicate definition, which simply calls the hooks and the invisible definition. From the outside a preprocessed predicate looks like a normal one, apart from the extra arguments (which a lower level implementation could provide implicitly).

Built-ins and other uninteresting (i.e., “predeclared correct” via specific rules in the custom oracle theory) predicates are not preprocessed, except for metapredicates and side-effects.

In order to support assert and retract, program clauses always have an associated creation time and optionally a destruction time, defining a time interval in which they exist.

In order to support builtins involving external side-effect (I/O)[1], these are always made complete access points, to avoid fiddling with the "external world" state while recomputing. This is implemented using "glue" predicates that are preprocessed.

7.4.2.2. Execution machine operations

The execution machine is used for 3 distinct operations, corresponding to different situations.

Top goal. Launch a top goal, making it a complete access point. More than one top goal can be browsed at once, since all information is kept associated to some (unique) time instant.

Complete information. Given a goal number, recompute its goal call (if not yet stored) and initial solution segment. Both the access point whose call is used to launch the recomputation and the access point looked for become complete.

Zoom-in. Given a goal number of a complete access point, recompute its goal call with smaller Wmax=0, in order to create more access points. I.e., refines an *abstract* computation representation into a *concrete* one.

We now describe the additional environment required for each goal call, during execution of the "top goal" operation. We then give a pseudo-code description of the processing done at the CALL port. The Prolog source code for the "top goal" operation is given in appendix G. For a pseudo-code description of the other ports and operations, see appendixes D,E,F.

The environment for each goal

In addition to Prolog's standard environment information, and to a a global "Time" counter, the HyperTracer uses the following variables for each active goal:

Flags

• Keep_times True iff the goal is (already) an ap.

---

[1] However, input side-effects are not supported by the *diagnosers*. They're simply tolerated.

• Keep_solutions True iff the goal is an ap for solutions

• Ilaunch True iff the ap has stored goal call, and therefore can be use to launch recomputations

• Father_is_launcher True iff the goal's father has Ilaunch true (passed as an argument)

• top_down_concrete True iff until now all goals immediately under this one have all solutions and failure facets stored. Passed as an argument to the goal's children.

Integers

• A reference to the nearest launcher (an ap with stored call) above, to access its local variables (passed as an argument)

• Last_enter Time of last REDO port for this goal

• Last_leave Time of last EXIT port for this goal

• Solution (order) number

• Previous ap facet in current derivation (actually an integer *pair*: one for the goal call, the other for the last goal solution)

• Father clause reference

• Nearest cut forward, if any (i.e., its clause instance match time)

Only if Ilaunch is true:

• saved_time Time that would be saved by the last computation fragment of this goal, were its coresponding solution stored and used as lemma in a recomputation.

• alien_time Time spent not under the goal, since the time of its CALL port; used to compute saved_time above.

Processing at the CALL port

Following is the pseudo-code description of the processing done at the CALL port, for the “top goal” execution machine operation. Processing for other ports has a similar flavor (cf. appendixes).

```
Keep_times=FALSE, Keep_solutions=FALSE
top_down_concrete (for my children!)=TRUE
```

```
Last_enter=time
If there are cuts in my father
      pass down the appropriate “cut” (match) ID
else
      pass down my father's
Saved_fragment_time=0
If the call is costly  || I must be an ap  (side-effects,
etc.)
      Creates new ap
      Store my (inherited) father_is_launcher flag
      saved_time=0 (time saved by using stored solutions)
      alien_time=0 (time spent not under this call)
      Stores the call
      Keep_times= TRUE
      Ilaunch=TRUE (I am a launcher)
      Pass down reference to this call as the launcher
      Pass down father_is_launcher=TRUE
      Store “before cut” info
      Store previous ap facet in derivation
      Store my father's clause reference
      pass down this call ap as previous facet
else
      Ilaunch=FALSE
      Pass down reference to old launcher
      Pass down father_is_launcher=FALSE
      pass down old previous facet
Increment time
```

The cost of recomputing a call or a solution from the closest stored ancestor (the launcher) is:

Cost = time of port (call/solution) - time of nearest launcher (may be THIS goal's call!) - alien_time of launcher - saved_time of launcher

7.4.3. Building suspect sets

Suspect sets are computed on demand from the (minimal) information kept in access points, and passed on to some diagnoser algorithm which later selects an interesting one to show the user.

For example, here's the code to obtain the **AND tree** for a goal solution - equivalent to its suspect tree, for definite and side-effect free Prolog programs:

```
abstract_and_tree(S,T,Next) :-
    abstract_and_tree_(S,TT,Next),
    reverse_tree(TT,T).

abstract_and_tree_(S,and_tree(S,Children),Next) :-
    previous_solution(S,PS),
    (
        is_under_solution(PS,S) ->
abstract_and_children_(S,PS,Children,Next)
            ;   Children=[], Next=PS
     ).

abstract_and_children_(A,S,[C1|Cn],Next) :-
    abstract_and_tree_(S,C1,P),
    (
        is_under_solution(P,A) -> abstract_and_children_(A,P,Cn,Next)
            ;   Cn=[], Next=P
     ).
```

The tree is built simply by walking through the linked list implicit in the “previous” relation, restricted to solution facets by the use of `previous_solution`, and taking notice of when solutions are under others (using `is_under_solution`, which simply checks computation fragment/solution times), hence imposing the tree structure. The obtained tree is reversed, to obtain the standard chronological order, from left to right.

As another example, here's the code to obtain the failure tree for a goal failure - equivalent to the **suspect tree for a goal solution set**, for definite and side-effect free Prolog programs:

```
failure_tree(Failure_facet,Tree) :-
    retractall(is_marked(_)),
    failure_tree_(Failure_facet,Tree).

failure_tree_(F,failure(F,Children)) :-
    (
        next_failure_below(F,The_F,_) ->
failure_tree_children_(F,The_F,Children)
```

```
            ;   Children=[]
     ).

failure_tree_children_(F,The_F,[C1|Cn]) :-
    failure_tree_(The_F,C1),
    (
        next_failure_below(F,FF,_) -> failure_tree_children_(F,FF,Cn)
            ;   Cn=[]
     ).

next_failure_below(FID,The_FID,APID) :-
    last_valid_failure(FID,The_FID,APID),
    assert(is_marked(The_FID)),
    !.
```

As failure facets are incorporated into the tree, they become marked (with a fact `is_marked`); `last_valid_failure` returns the latest solution set (failure) facet under the current ap which is not yet marked. Taking the latest failure also guarantees that it is the closest to the current one (i.e., such that there's no intermediate goal with stored failure facet).

Suspect trees for impure programs, namely with **cuts**, are built similarly, using mutual recursive calls to variants of the two previous predicates, and taking into account the information in the `before_cut` relation in the execution database. Suspect trees for **output segments** are built similarly to failure trees, and using the information in the `a_side_effect` relation of the execution database.

The current version of the HyperTracer does *not* build extended suspect trees taking into account **internal database side-effects** (as defined in chapter 4). Before showing a diagnosis, it simply checks whether it was dynamically produced, and if so, gives a warning and immediately continues with the suspect tree for the database side-effect.

7.4.4. Diagnosis algorithms

To avoid modal (rigid) user interactions, the diagnosis algorithms are used incrementally. When the user gives a WHY command the following steps are taken:

• The suspect tree for the selected goal behavior facet is built from the Execution Database, as explained in the previous section.

• It is refined according to the current oracle knowledge, by traversing the suspect tree checking for nodes stated correct by the oracle.

• It is passed on to the appropriate diagnosis algorithm, which chooses a goal facet to be visited (inspected) by the user, eventually a source text chunk - a diagnosis.

• The chosen goal behavior facet is shown and selected, ready for further user action on it: either continue with another iteration of the (present or other) declarative algorithm, or simply browse around.

The AD&Q and N&Q diagnosis algorithms are implemented each using two tree traversals: the first to count nodes and different components in the suspect tree, resp.; the second to actually choose the best node.

SECURE is implemented as a variant of N&Q: N&Q simply checks whether the SECURE assumption should be used, and checks also if it becames invalid, turning it off automatically after notifying the user.

Declarative source debugging (N&Q and SECURE algorithms) is available only if **the weight bound of the execution machine is 0**. Although a “diagnosis” found in an abstract suspect set by a declarative execution debugging method is useful, (because a subsequent concrete diagnosis can be found on the subtree to be later “zoomed-in”), it is useless for declarative source debugging: the goals which were ignored, because they aren't stored access points, may use program components absent from the “abstract source suspect set”, and the diagnosis may be among them.

7.5. The “HyperInterface”

In order to implement the HyperTracer interface, it became necessary to follow an object-oriented methodology, to couple with its complexity. The “HyperInterface” is the HyperTracer subsystem implementing such a methodology. It assumes that a simple Graphical User Interface is available, simply providing for the creation of graphical objects producing distinct events, and provides class mechanisms on top of it, implemented in Prolog. In principle, the mechanisms it provides could be useful to other programs as well.

7.5.1.Motivation

Making a good human interface is hard work. The HyperTracer's interface is an example, since it is expected to render on a graphical screen Prolog goals, solutions, execution trees, source programs, etc., all in a consistent and integrated manner, although using an Prolog environment already with good graphic interface primitives. The effort spent on a older HyperTracer version with its interface (around 47 k of Prolog source) made us rethink it, leading us to the HyperInterface (HI) interface subsystem.

The current HyperTracer interface is implemented still with around 30k of debugger-specific, but provides a larger functionality, is clearer, and benefits from several new generic interface features. Its cosmetics is poor, because it was not a design objective: functionality, flexibility and portability were the objectives - cosmetics (in particular, additional graphical atributes, automatic layout methods, etc.) can be added a posteriori, and depend on each particular Graphical User Interface.

The HI itself consists of about 65k of Prolog code, independent of the particular application using it (in our case the HyperTracer), and like the HyperTracer is currently running on the MacLogic environment [3], on Apple Macintosh computers, and on the ALPES Prolog environment [2], on X-Windows workstations.

7.5.2.Overview

The HI is asked by the Prolog program - in our case, the HyperTracer - to create, destroy or update graphical *objects*, and lets the user interact with them via *commands* (cf. below). There are higher-level primitives available to create several objects at once using an arbitrary (binary) Prolog relation.

The objects are instances of classes, organized in a single hierarchy, each class consisting of a sequence of <event, condition, action> clauses (“methods”). Whenever the user produces an event (e.g., doubleclick on a list item), it is first mapped into a higher level event by the underlying graphical interface layer (i.e., MacLogic's), and is then fed to the currently selected object. From there it may be propagated to the class hierarchy above if necessary, until a class can deal with it, by accepting the event while verifying the condition, and executing the action goal.

An object is either a *window* as a whole, or one of its *parts*. On the other hand, each object is an instance of a HI *class*, and has an *identifier*. Each object can therefore be uniquely referred to by either a pair <Window,Part> (the HI internal representation) or by a pair <Class,Identifier> (the application representation).

Objects also have a *name*, which can be regarded as a user-readable version of the identifier. When the name is considered too big by the HI, it is shown in abbreviated form. There are several commands to display the full name of an object in different ways.

Let's look at an HyperTracer debugger example:

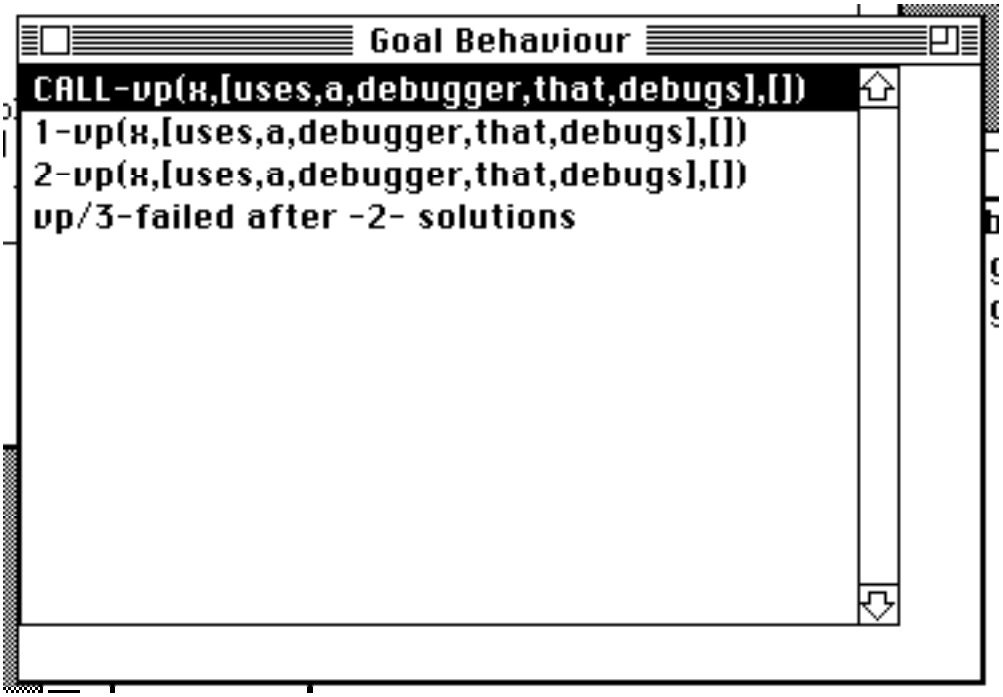

Figure 7.3: An HyperInterface object

The selected object has name " CALL - vp(x, [uses, a, debugger, that, debugs], []) ", class gb_call_item, identifier 596 (an application - HyperTracer - reference to the goal call), window 6111236 (the internal MacLogic window reference) and part 1 (the first item in a list window).

A *command* is a user event, such as a pull-down menu command or a double-click on a dialog list item, and always refers to some selected object. Each object has an associated set of usable commands, *implicitly*[1] specified by its class. For example, the pull-down menu associated with a window[2] contains only items relevant to the window and its parts.

Application actions, namely those with consequence on the graphical interface, are strongly determined by the previous user command. Whenever an HI object B is created by the application as a response (immediately after) an user command applied to selected object A, we say that "A *created* B", or that "B *is created by* A", or that B is one of A's "*creations*". This "creation" information is crucial to some HI features: object creations typically have a strong causal connection to their creator, from the application's semantics point of view, irrespective of the application at hand.

The HI keeps some session information independently of the application, such as the object creation relation above, event logging, etc. This, plus the taking of some reasonable assumptions, allows the HI to provide some fancy default features, therefore avoiding the need to program them explicitly. For example: updating groups of objects, going back to the last object or showing the object which "created" the current one using a menu command, or selecting a "created" object with a double click, are all default features provided independently of the application.

The "HyperInterface" designation comes mostly from these features. Hypertext-like links among nodes are established implicitly by the actions taken in class methods, be it derived from different menu commands, doubleclicking, etc.: the "object creation" relation, which the HI maintains automatically, is adopted by default as the explicit link.

Finally, there's an on-line help system with browsing facilities, associating an (editable) chunk of text to either a class or a command for a particular class, and also a command macro recorder facility.

---

[1]Menus are automatically generated, using a simplified lookahead technique based on [15].

[2]The menu bar is updated transparently by MacLogic, who knows which menus belong to which windows.

7.5.3. Implementation

The HyperInterface is part of the HyperTracer, and its native version runs on the MacLogic/C-Prolog. As stated above, there's also a version running on ALPES-Prolog. There's still another (incomplete) version running on HyperProlog/YAP, an HyperCard[1]-based programming environment developed by the author.

7.5.3.1. Basics

An object is uniquely identified as follows:

• from the HI side, by a pair <window reference, window part>;

• from the application side, by a pair <Type, Identifier> (type meaning “class”).

Events currently have the following format:

```
my_event(
        MacLogic_event, Object_ID, Window+Part,
        my_modifiers(Shift, Option, Command, CapsLock),
        Type
        )
```

The <Type,ID>, <Window,Part> refer to the (same) selected object, to which the event is adressed.

7.5.3.2. Types, their hierarchy and event-handling

A type/class is defined by a Prolog predicate. Each clause represents a method. For a type named foo_type:

```
foo_type(Event,Action) :- Condition.
```

The only restriction is that menu events should be fully explicit in the clause head (i.e., be ground). This allows automatic menu precompilation (cf. below), using a simple lookahead technique.

The hierarchy is defined by the subtype_of/2 predicate, whose arguments are type names. Event handling is basically done with some meta-programming:

---

[1] HyperCard is a multimedia application toolkit, from Apple Computer.

```
dispatch_event(My_event,Type) :-
      functor(Type_template,Type,2),
      arg(1,Type_template,My_event),
      call(Type_template),
      % Event matching and Condition testing…
      !,
      % this type will handle the event
      arg(2,Type_template,Action),
      call(Action). % now having arguments bound by condition

dispatch_event(My_event,Type) :-
      % otherwise some supertype should do it:
      supertype(Type,Super_type),
      !,
      % deterministic: the type hierarchy is a tree
      dispatch_event(My_event,Super_type).
```

Event types depend on the underlying environment. They should contain at this stage a unique identification of the selected object and the event itself (menu command, doubleclick on object, etc.). As a matter of fact they need not be related to user events - for example, the HI uses special events (or messages) to create and destroy nodes, inquire object names, etc., and the Prolog application can use its own as well.

The MacLogic version uses only the following events: doubleclick on a node, optionally complemented with modifier keys, and pull-down menu command. We decided to keep the interface cosmetics as simple as possible, to make it more portable to other environments. For example, popup menus were left out.

7.5.3.3. Internal HI structures

The main data structures used by the HyperInterface are:

• A relation of interface objects: browser_node_instance(Type,ID,window,Part)

• A relation of user events: user_event(Chronological_number,event)

• The “object creation” relation referred above: created_node(Window1, Part1, W2, P2, Event_number)

• A relation of menus for each object class. Pull-down menus are built dynamically for each class, by looking at the events the class accepts. Using this information a relation type_menus(Type,Menu_list) is built.

• Class hierarchy relation. The class hierarchy information, specified by the user with facts "class_name(Super_class,is_my_supertype)" near the class definitions, is made available via the following relations:

supertype(Type,Super)

subtype_of(Type,Supertype)

7.6. Performance and future implementations

The current HyperTracer is implemented in Prolog, as a combination of preprocessor plus runtime predicates. The preprocessor simply inserts some calls ("hooks") to runtime predicates implementing the required port processing (cf. comments at the beginning of the listing in appendix G). This approach proved effective for basic experimentation, because it is flexible and portable, but it is irrealistic for a practical debugger.

Implementing the port operations in Prolog introduces too much overhead, as illustrated in appendix G. The current prototype, running on a C-Prolog interpreter on an Apple Macintosh IIx computer, has a speed of about 10 lips for debugged programs (instead of about 1500 lips for normal execution!). We compiled the HyperTracer (and the preprocessed programs being debugged) with a good Prolog compiler, and the result was just a 4 time increase in speed.

There's however an important aspect to consider: the time and space overheads for executing non-access point goals are *constant and independent of the program* [1]. We'll now estimate the space and time overheads for an ideal implementation, at the Prolog machine implementation level.

The following table was obtained by counting basic operations in the HyperTracer (HT) execution port algorithms, and those performed by a standard compiled Prolog implementation[2] to execute an iteration in the append predicate (part of the standard "naïve reverse" benchmark). The time costs for the ideal HyperTracer assume that the "hook" calls to the HyperTracer ports are inserted by the Prolog compiler, as WAM instructions.

---

[1] This was a major implementation design objective, satisfied only on the latest HyperTracer described here.

[2] SB-Prolog 3.0, a Warren Abstract Machine variant, from the University of Stony-Brook, NY, USA.

Cost per goal:	Time	Space in stacks	Space in DB
Standard execution	25 assignments 17 conditional jumps 8 jumps	One new list element (if goal needs an environment, 2 words plus local variables; if goal has a choice-point, additional 6 words)	0
HT, non-access point goals	52 assignments 28 conditional jumps 18 jumps	10 words[1]	0
HT, complete access point goals (N solutions)	74 assignments 33 conditional jumps 18 jumps Time of storing N+1 terms	10 words	6+N words N+1 terms (call and solutions)

Table 7.2: Execution overhead estimates

As can be seen above, the ideal HyperTracer would introduce a speed overhead a bit over 100% for non-ap goals, and over 200% for access point goals (depending on the number of solutions and size of goal terms), on a typical Prolog implementation. The stack overhead would also be acceptable.

The HyperTracer weight bound W_{MAX} (cf. "Choosing access nodes" section above) could thus be made large, avoiding in practice the solution storage problems refered above (cf."Time vs space trade-off" section): assuming a Prolog performance of 40Klips, already realistic in low-end workstations, to guarantee a maximum recomputation time of 1/10 second W_{MAX} should be set to 40000/10/2=2000, well above branching factors of most Prolog execution trees.

[1] Memory words are typically 4 bytes long.

In conclusion, *we conjecture that the virtual trace storage mechanism is worth a serious implementation attempt*.

To support the declarative source debugging methods diretly in the execution machine, we would need O(C) space *for each execution node*, where C is the number of program components. In general, for each execution node (i.e., each goal behavior facet in the execution tree) we would need to keep its own information regarding the use of all program components. Although the use of program analysis can minimize this cost (say by associating predefined and smaller data structures with each textual goal call in the program, according to the information in a "A calls B" graph), it won't change its character.

This suggests either to change the declarative source debugging methods, to simplify their implementation[1], or to delay all source debugging - related proccessing to the debugging phase (i.e. after the execution terminated, with all goals stored as access points, by setting W_{MAX} to 0). We followed the latter approach in the HyperTracer.

[1] For example, a simplified version of SECURE could be as follows: during execution, and for each program component, keep only the K first goal behavior facets that match it; then apply algorithm SECURE, using only this partial information. This suggests a very efficient implementation, but whose usefulness we're unable to quantify at this point.

Part III: Cross-fertilizations

8. Debugging and Knowledge Base Updating

In this chapter we concentrate on the problem of updating knowledge bases (KB), assumed realized as normal logic programs, using negation as failure. We introduce a new KB update method inspired on and using the present debugging framework.

The material in this chapter has been presented in [71], and resulted from work in the ESPRIT BRA COMPULOG project.

8.1. The problem

The problem we want to solve is the following: given a normal program P and an update request, find a set of candidate transactions, and apply one to obtain a new program P' that satisfies the update [1].

An *update request* is either of `insert G` or `delete G`, where G is a goal; insert G is a request to transform P so that G becomes provable; delete G is a request to transform P so that G becomes false (unprovable, given the use of negation by failure).

A *transaction* is a sequence of program edition *actions*; these can be either retract(C) or assert(C), where C is a clause to be removed or added to P, resp. A *candidate transaction* is one of several transactions which, if performed on P, will transform it into a program P' that *may* satisfy the update request.

8.2. Outline of the solution

Our method is inspired on the following analogies:

KB updating	**Debugging**
Current KB	Buggy program
KB satisfying the update request	Correct program
Update process	Debugging+bug correction
insert A	A is a missing solution - find the bug and correct it
delete A	A is a wrong solution - find the bug and correct it
Update request information, additional specification of the desired theory[1]	Oracle knowledge

Table 8.1: Knowledge Base updating vs Debugging

The idea is to proceed as follows, given an update request R:

• Find a **suspect set** for the goal behavior facet corresponding to the update request, by simulating its execution.

• **Generate candidate transactions from the suspect set**; if none could be generated, backtrack to the previous iteration, and resume after choosing a different transaction in the step below

• **Choose and apply a *compatible* transaction** to the program.

• If the program still doesn't satisfy the update request, **iterate** the procedure from the first step, with the transformed program.

The suspect set (cf. chapter 2) to consider for each update request follows:

• For a “delete G” request, SS(solution(G))

• For an “insert G” request, SS(solution_set(G))

In the present context, two actions *contradict* each other if they have the forms assert(F1) and retract(F2), and F1=F2 modulo variable renaming. It is *not* necessary to perform a subsumption test between F1 and F2[1].

---

[1] For example “annotations” on which relations to change [87] can be regarded as oracle knowledge.

A *compatible transaction* is one such that neither of its actions directly *contradicts* another, in the whole transaction accumulated along the successive iterations, nor has a variant (i.e., identical to it modulo variable renaming) already in the transaction. This restriction prevents nontermination problems from originating in the iterative process itself.

8.3. Generating transactions from suspect sets

Having obtained suspect sets for the computation corresponding to an update request, how can we obtain candidate transactions ?

8.3.1. The “suspect subset” transaction generator (SSTG)

Let SS' be the suspect set to consider for the update request, as defined above (if more than one suspect set exists, such as in a solution with different proofs, consider all the respective suspect sets). The set of all candidate transactions is defined as follows:

• Consider all nonempty subsets S' of SS'.

• Each ordering of S' corresponds to one candidate transaction, with a transaction action for each S' element, as follows: for each clause instance C, an action retract(C); for each predicate definition instance <G,P>, an action assert(G).

• The following restrictions apply to any transaction, thereby eliminating some candidate transactions:

- if it contains an action assert(G), then it cannot contain any action involving suspects matching goals under G, nor can it contain actions involving goals not in the same SLDNF-tree path as G;

- if it contains an action retract(C), then it can't contain any action involving goals above the goal matching C.

In both cases the eliminated actions would be redundant.

---

[1]If an action “contradicts” another because its argument subsumes another's, a later contradiction conflict, in the sense above, will occur in a subsequent iteration of the update process.

Notice that we define transactions by combining members from a suspect set, corresponding to nodes on an AND/OR tree, instead of starting directly from nodes in an SLDNF tree, as was done in [38].

Example Consider the following program, borrowed from [38], and the update request `insert pleasant(fred).`

```
pleasant(X) :- ~old(X), likes_fun(X).
pleasant(X) :- sports_person(X), loves_nature(X).
sports_person(X) :- swimmer(X).
sports_person(X) :- ~sedentary(X).
old(X) :- age(X,Y), Y>55.
swimmer(fred).
age(fred,60).
```

The suspect set is:

SS(solution_set(pleasant(fred)) = {age(fred,60), old(fred), < loves_nature(fred), loves_nature/1 >, < pleasant(fred), pleasant/1 > } →

Notice that neither <swimmer(fred), swimmer/1> nor <sports_person(fred), sports_person/1> is in the set, because of the statement subsumption rule in the oracle theory (cf. “Customized Oracle Theory ”, chapter 2). The arithmetic predicate is assumed to be “non-updatable”, or correct, in debugging terminology.

Given the above restrictions filtering candidate transactions, the candidate transactions are only:

```
[retract( old(X):-age(X,Y),Y>55 )]

[retract( age(fred,60) )]

[assert( loves_nature(fred) )]

[assert( pleasant(fred) )]
```

These are only some of the transactions obtained with the method of [38], which includes transactions with redundant actions.

8.3.2. Relationship to other methods

We'll now examine how the SSTG compares with other transaction generators, without giving special attention to the effects of integrity contraints considered by other authors. These can be seen as part of the update request, both in a SSTG-based and in other methods, and so we'll be assuming an empty integrity constraint theory. Or, if you will, we'll be comparing the "possible transaction" generation capabilities of the methods, irrespective of whether they satisfy the integrity contraints.

Comparison with Guessoum and Lloyd's method

We refer to the [38] transaction generator as "GLTG".

The GLTG method is mutually recursive due to negation (the deletion method calls the insert method, and vice-versa). This recursion is implicit in our definition of suspect set, which is also "mutually recursive" regarding the suspect types, also because of negation.

Retract actions, and their contribution to the candidate transactions, are the same in both methods. This is not the case however for assert actions:

• The GLTG produces one assert action for each atomic goal in a definite goal in the SLDNF tree, even if such atomic goals are never activated (because some left brother fails).

• The SSTG produces one assert action for each atomic goal that failed, and which therefore was the leftmost goal in some goal node in the SLDNF tree. It does *not* produce assert actions for atomic goals which were not activated, i.e., were never matched against clauses in the program.

• The SSTG imposes the restriction that for all assert actions corresponding to a MSS subset, none is an ancestor of another. The same restriction is implicit in the GLTG: given a definite goal in a SLDNF tree, it is never the case that one of its atomic goals is an ancestor of another.

Conclusion: our SSTG depends on its iterative nature to (later) generate the missing assert actions if they're relevant, corresponding to atomic goals which were not activated.

Comparison with Decker's method

Decker [21] created a transaction generation method akin to Lloyd's, except that he uses a special selection function for SLDNF execution, defining *view update trees*.

Our SSTG is defined based on suspect trees, which are defined from SLDNF-trees. It should be easy to define suspect trees and sets from "view update trees", and thus build an SSTG over it. We regard "view update trees" as the result of yet another execution mechanism, and therefore as an independent issue, just as if we were using (say) sidetracking or intelligent backtracking (cf. Section "Clever execution interpreters", Chapter 2).

Given the similarity between Decker's and Guessoum&Lloyd's transaction generation methods (modulo the different "execution mechanisms" they use), we conjecture that the same comments apply.

8.4. A KBUM prototype

We implemented a prototype following the outline above, and based on generation of just singleton transactions. It is nicknamed **K**nowledge **B**ase **U**pdating **M**achinery, and its algorithm is:

• Given an update request, check whether it is satisfied, and terminate if so, giving as a result the concatenation of all intermediate transactions.

• If it is not satisfied, generate the candidate singleton transactions (i.e., with a single action) from the appropriate suspect set using the SSTG, choose[1] an arbitrary but compatible transaction, apply it (hypothetically) to the program, and iterate this procedure. Whenever no non-conflicting transaction can be generated, backtrack and choose a different candidate transaction.

At this stage we've implemented KBUM with some simplifications, namely ignoring other than the first solution of G in ~G failures, with simplistic criteria for choosing transactions, and without infinite derivation checking.

---

[1] See "Choosing among transactions" and "Update space searching" sections in [71] for comments on how to choose.

The current prototype is based on an interpreter and auxiliary predicates written in Prolog, which can be found in appendix H. “Hypothetical changes” are simply “performed” by adding actions to a pair of lists carried along, containing the performed assert and retract actions. The interpreter produces a partial suspect tree for each solution, and, on failure, leaves asserted information sufficient to easily build suspect trees for failures (solution sets) and to complete the suspect trees for solutions. Each algorithm iteration corresponds to an execution by the interpreter.

Although the experience with the HyperTracer was crucial in developing this prototype, we didn't use the HyperTracer execution machine, whose development proceeded in parallel.

Example Here's another update problem (example 3 from [21], section 3.7) : inserting p in the following normal program:

```
p <- r,~q.
q <- r,~s.
```

And here's KBUM's output:

```
?- insert(p,T).

T = [assert(r),assert(s)] ;

T = [assert(r),retract_action(q, #<80005ed3>)]  % reference of
clause

T = [assert(r),assert(p)] ;

T = [assert(p)]

 →
```

Part IV: Conclusion

9. Conclusion

In this final chapter, previous work on logic program debugging is overviewed from the perspective of the present framework, and a list of research problems and work opportunities is given.

9.1. Relationship to other logic program debugging work

In order to better view what's added by the present framework, we'll for some authors use a graph, picturing a “territory” of possible approaches, in complement to a brief textual description of differences. We found out that not all dimensions of the territory are amenable to a natural graphical representation (at least by this author), and thus authors operating in those regions are treated just textually. No offense intended!

First, the graph showing **what's covered by the present framework** in terms of the sequencial logic programming language features; parallelism issues will be referred to explicitly for those authors dealing with them.

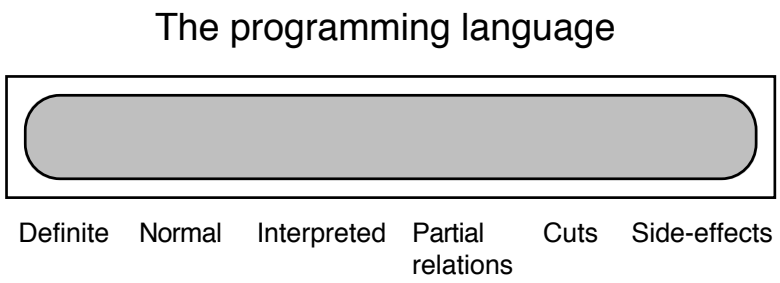

Figure 9.1: Our language territory

A region with the pattern  means that the present framework covers it.

Next, another graph, showing the types of bug manifestation (error symptoms) treated, and the diagnosis algorithms available for each. A region with pattern  is covered only partially by the framework:

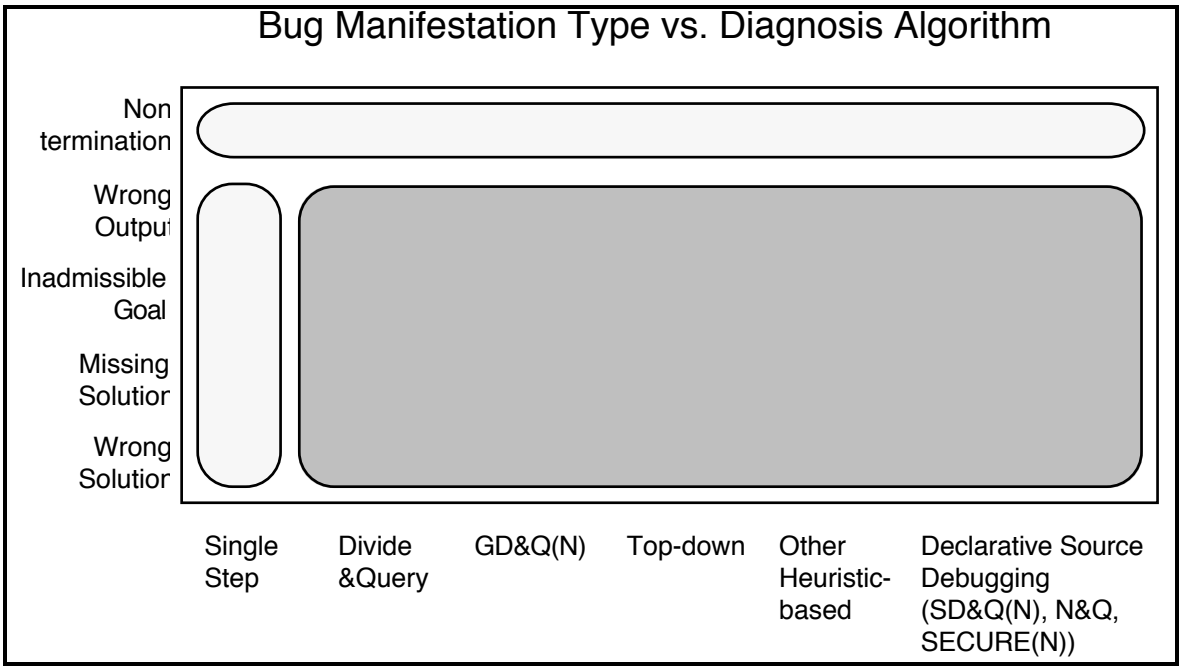

Figure 9.2: Our algorithms

In our case, non-termination symptoms can be treated by interrupting execution and applying the diagnoser to the partial computation, thereby assuming that the last goal call is inadmissible, or after manually browsing through the computation and finding another bug manifestation; however it cannot be guaranteed that a diagnosis is always found. Also, single-stepping is partially supported in the sense that the HyperTracer has manual browsing commands, but there's no declarative diagnoser using this algorithm.

Finally, we'll use a simple graph to picture the implementation approach, regarding the use of time and space when the access to failed execution tree branches requires either recomputing or storing them (in particular for treating missing solutions). For the present framework, the virtual trace storage mechanism particularizes into any of the two simpler approaches:

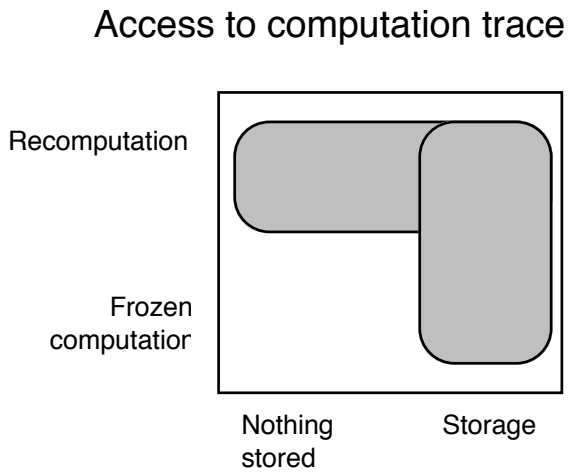

Figure 9.3: Our implementation

We group previous work into declarative and non-declarative sections, somehow arbitrarly: most of the (primarly) non-declarative systems mentioned also support declarative diagnosers.

9.1.1. Declarative Debugging

Apart from what's visible in the graphs below and above, there are a few very important differences regarding the present framework:

• **"Frozen execution"**; our diagnosers are defined over a frozen execution trace, thus avoiding the nontermination/incompleteness problems suffered by other diagnosers, and gaining flexibility: algorithms which are amalgamated with the interpreter executing the program[1] are necessarily less modular and so more difficult to change.

• **Suspect trees**; a related issue is our use of suspect trees as an uniform entity to define diagnosis algorithms, *independently of the programming language and its execution mechanism*.

• **Yes/No answers**; most authors require the user to provide also complete goal solution sets in some situations, or run into some problems when the user doesn't, because they don't work on (frozen) suspect trees.

• Integration in a **graphical interactive environment**; all other declarative debugging systems are text-based, with the exception of the Transparent Prolog Machine (cf. below).

---

[1] Like those of all other authors overviewed in *this* section.

Shapiro's “Algorithmic Program Debugging”

In his PhD thesis, Ehud Shapiro[83] invented algorithmic/declarative debugging, in the full sense of diagnosis+bug correction (in this work we concern ourselves only with improving the first part), and in the context of Prolog, but with emphasis on definite programs.

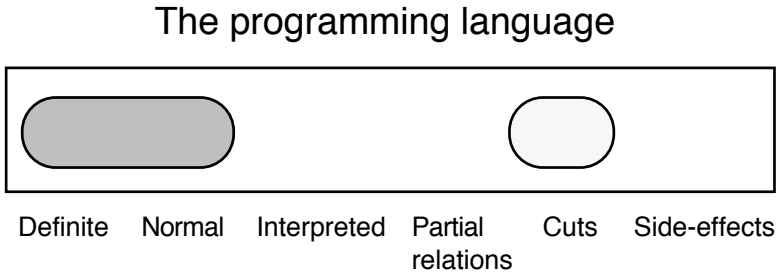

Figure 9.4: Shapiro's language territory

His treatment of cuts is however incomplete (cf. for instance our example in section “Bug instances with cuts”, chapter 4).

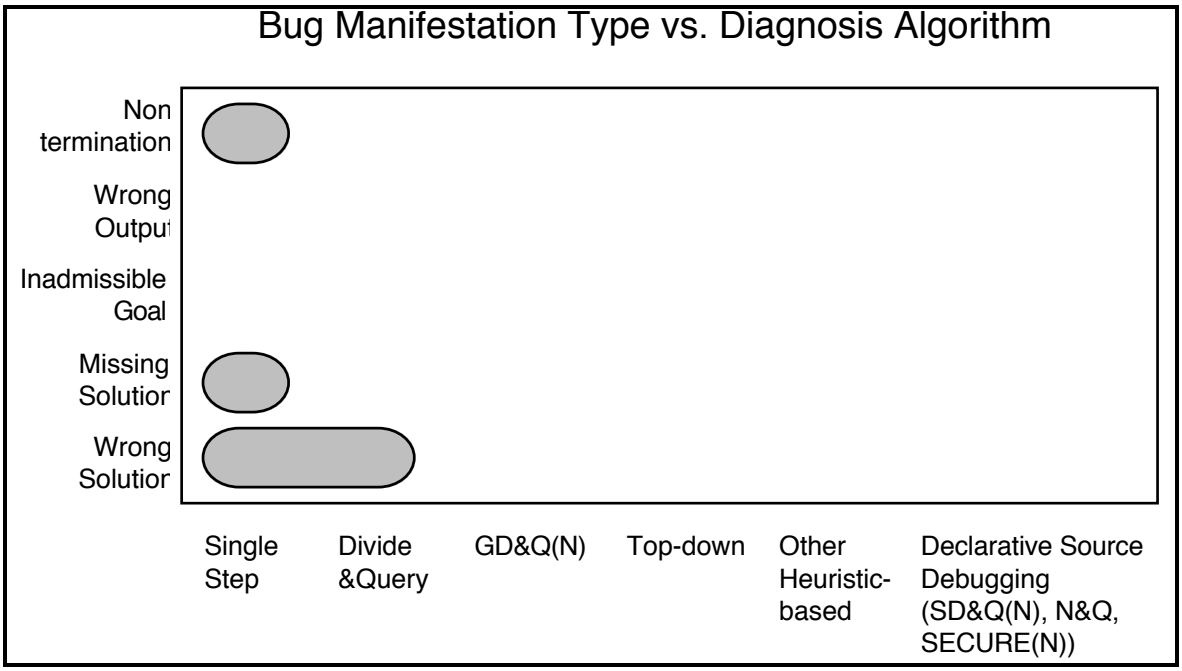

Figure 9.5: Shapiro's algorithms

He created the earliest diagnosis algorithms, and implemented them amalgamated with the meta-interpreter executing the program:

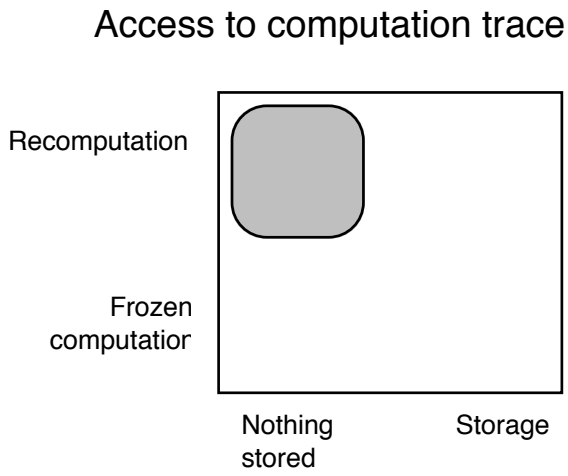

Figure 9.6: Shapiro's implementation

Plaistead's “Efficient Bug Location Algorithm”

In [76] an improvement is made over Shapiro's Divide and Query, by “forcing” the suspect tree to be binary, thus avoiding Divide and Query's problems with non-uniform trees. This is done by querying the user about the correctness of solutions with incomplete bindings[1] - which is equivalent to querying about conjunctions of clause subgoals - in such a way as to implicitly define a binary suspect tree.

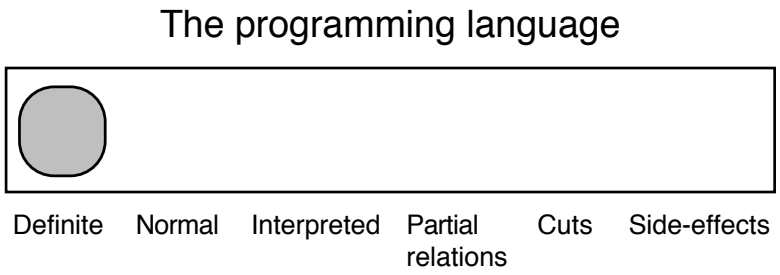

Figure 9.7: Plaistead's language territory

[1] A generalization of [64]'s “solvable” query.

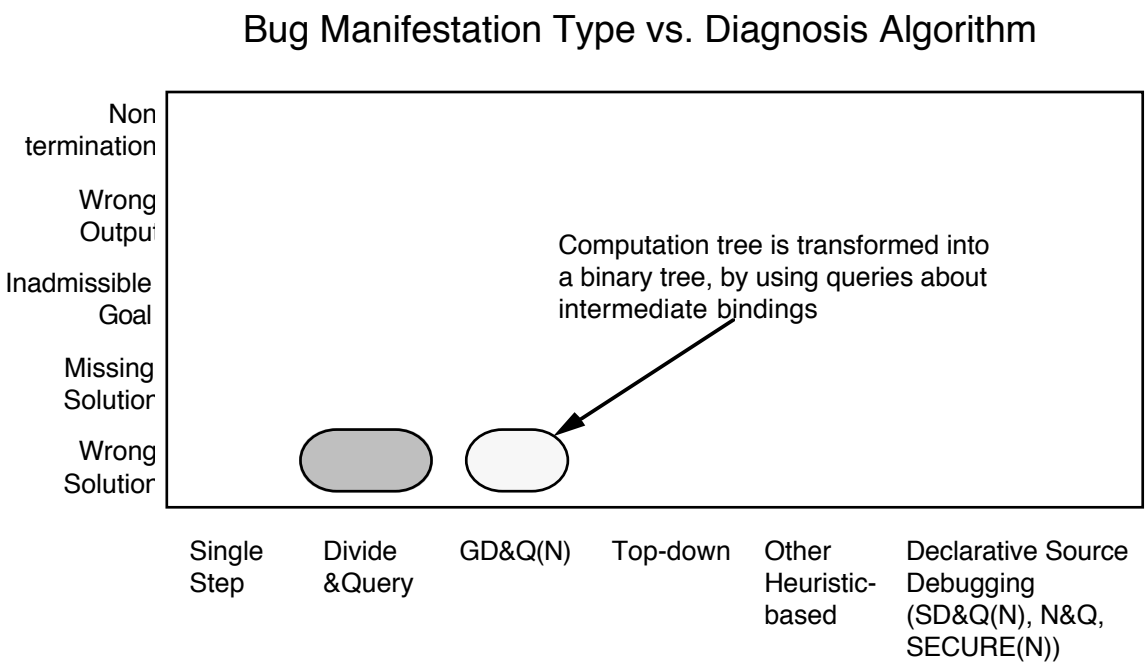

Figure 9.8: Plaistead's algorithms

The "binarization" of the computation is used also to support the first hybrid recomputation/storage approach. The idea is akeen to our own (cf. "Virtual trace storage" section, chapter 7), except that the fact of the trees always being binary simplifies the implementation.

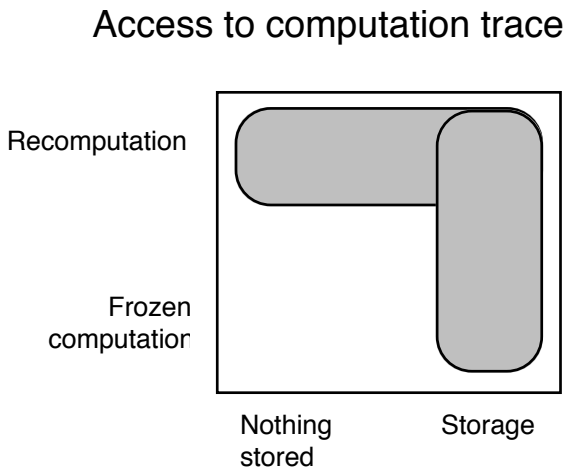

Figure 9.9: Plaistead's implementation

L.M.Pereira's "Rational Debugging"

Luís Moniz Pereira introduced a new approach [64], comprising the use of heuristics supported on knowledge of binding dependencies, and of intelligent

backtracking. The main motivation was both improving the diagnosis algorithms and extending the scope of declarative debugging.

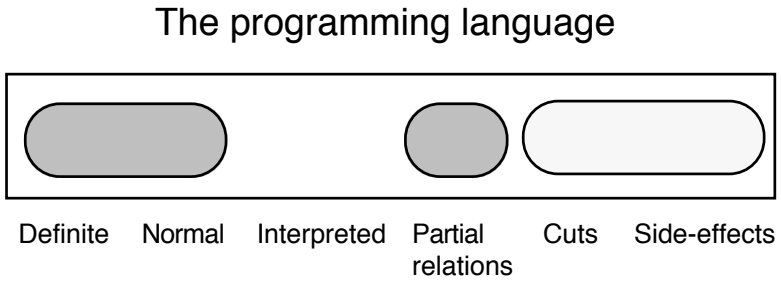

Figure 9.10: Pereira's language territory

Prolog's impurities were first tackled. Cuts and side-effects are tollerated in programs, but the diagnoser essencially ignores them, apart from giving warning messages in some relevant situations.

Binding dependencies are first used to allow the oracle a more informative answer level, by pointing at some particular wrong term within a goal solution, letting the debugger successively pursue goal matches on which those wrong terms' bindings depend. This, and other heuristics used, mechanize typical programmer debugging guidelines (such as “where does this term come from”), and address the lowering of the average number of questions, rather than minimizing the worst case.

Term's binding dependencies are also used to intelligently backtrack [75] over irrelevant subtrees, thus avoiding unnecessary questions within the method for missing solutions. The Rational Debugger is still the only implemented declarative debugger using intelligent backtracking.

A fourth type of bug manifestation was introduced, based on the concept of *inadmissible* goal, dealing with partial relations, and again using binding dependencies to detect the origin of error. Another type of query was also introduced: whether a goal is “solvable”, i.e. whether a goal partially instanciated after matching a clause can still produce a valid binding. And the user is not required to produce the correct solution instances (which Shapiro's missing solution algorithm required).

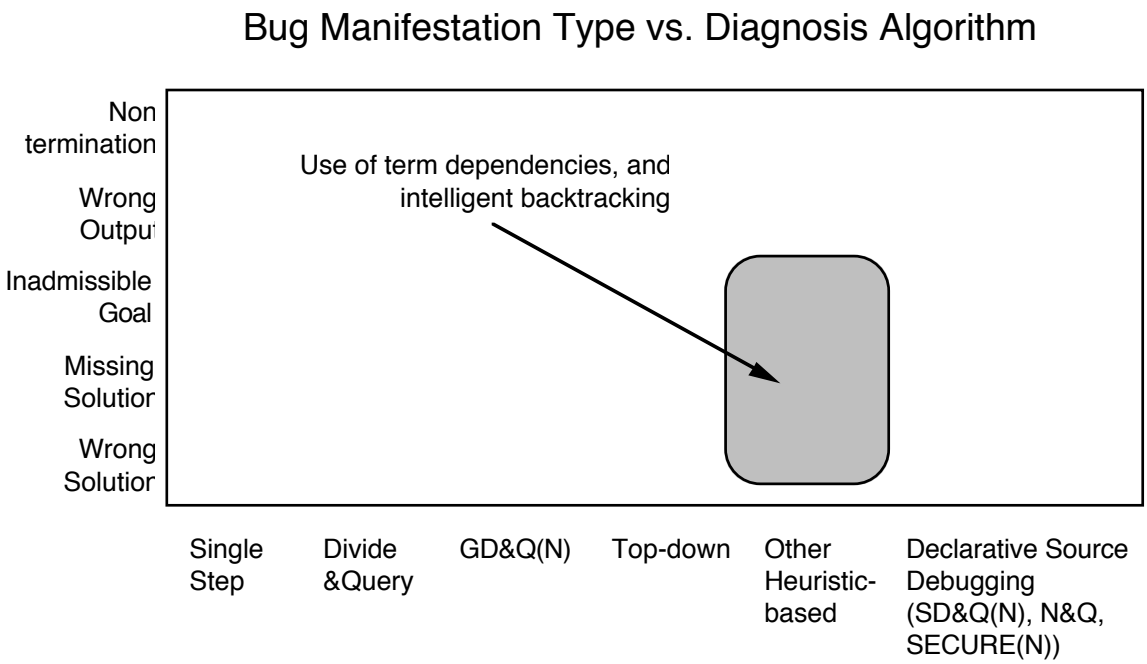

Figure 9.11: Pereira's algorithms

The implementation is based on a meta-interpreter, using intelligent backtracking. Side-effect calls are logged and can be undone whenever the diagnosing algorithms require it, through the use of special Prolog stream built-ins that were implemented for C-Prolog[61].

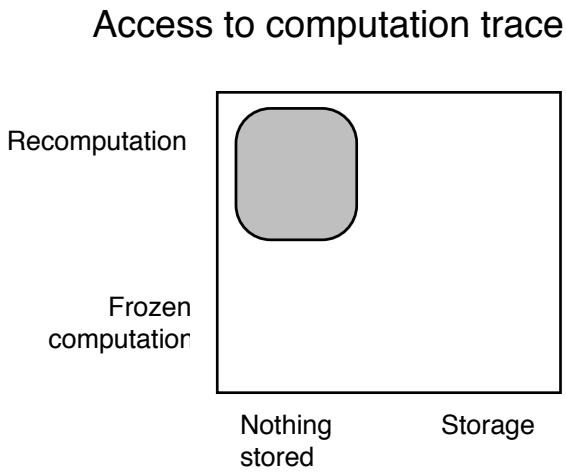

Figure 9.12: Pereira's implementation

“Top-Down Diagnosis”

Evyatar Av-Ron's MSc thesis[4] introduced the first top-down diagnoser, and also used wrong term dependencies, following Pereira's work.

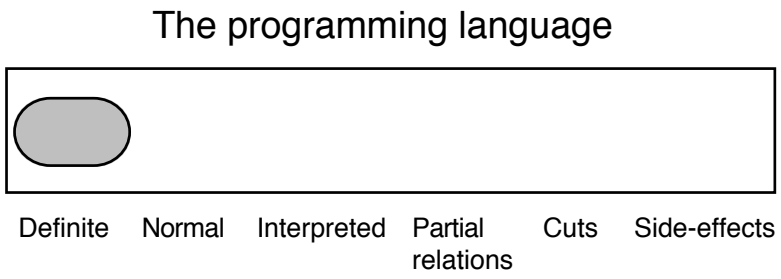

Figure 9,13: Av-Ron's language territory

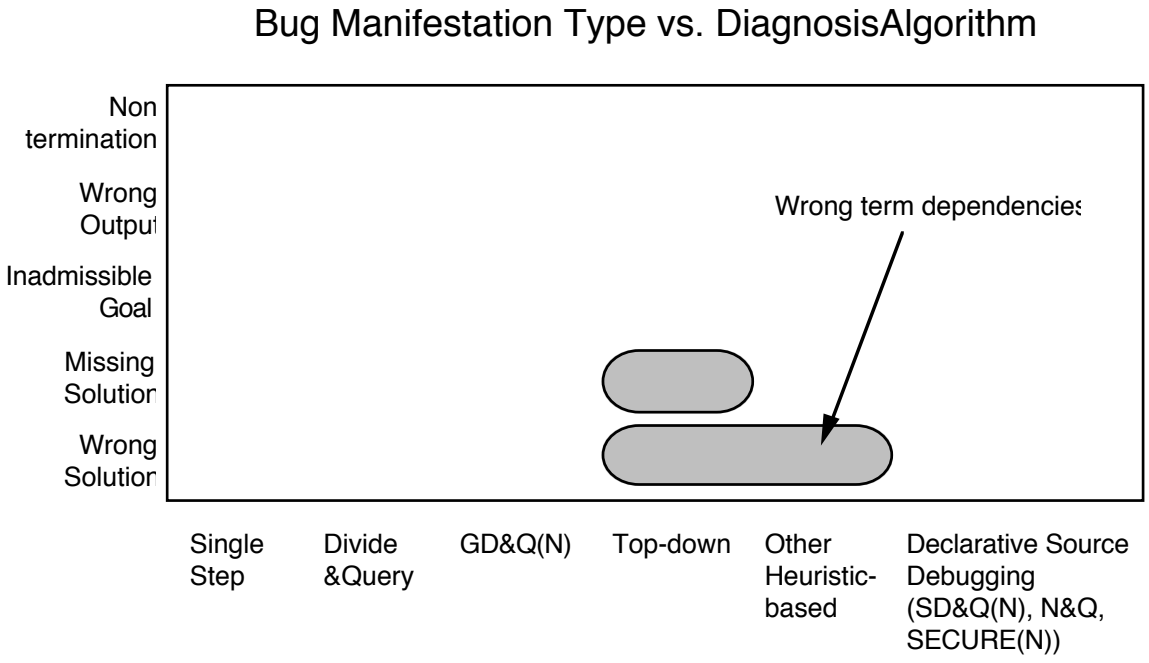

Figure 9.14: Av-Ron's algorithms

The implementation was also based on a meta-interpreter, recomputing suspect trees for missing solutions.

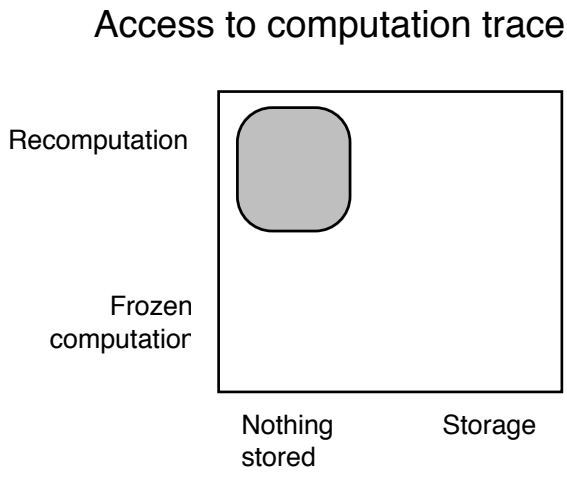

Figure 9.15: Av-Ron's implementation

K. Y. Koh [44] also developed a top-down diagnoser for wrong solutions, defined over a stored trace.T. Kanamori and others [43] developed a system using top-down diagnosis over a stored trace, explicitly integrated into a specification-testing-debugging cycle.

Lloyd's “Declarative Error Diagnosis”

In [48] John Lloyd applies declarative debugging to logic programs with arbitrary formulas in the body.

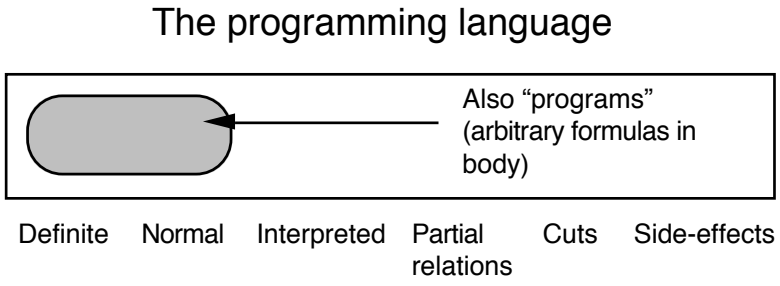

Figure 9.16: Lloyd's language territory

He also defined top-down algorithms for wrong and missing solutions.

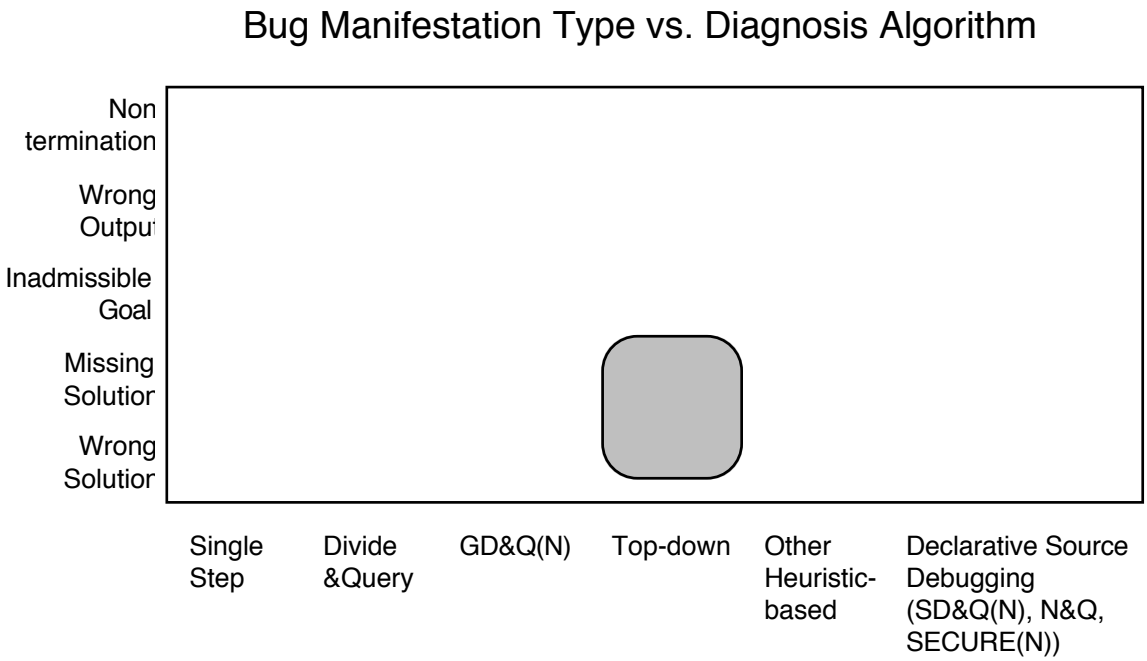

Figure 9.17: Lloyd's algorithms

His diagnosers are defined in a meta-interpreter.

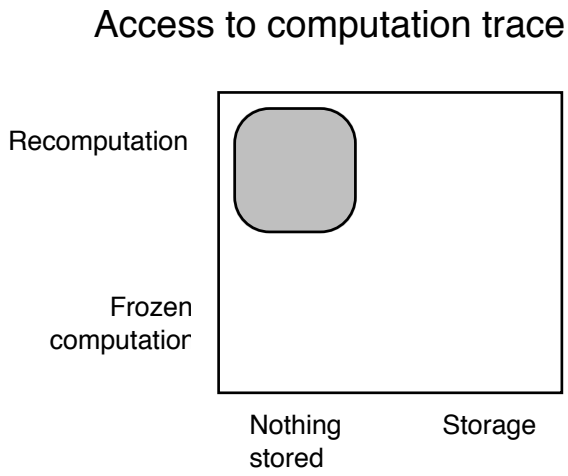

Figure 9.18: Lloyd's implementation

Naish: Missing solution improvements, NUDE

In [57], Lee Naish improves Lloyd's missing solution diagnoser regarding incompleteness, and compares previous missing solution algorithms. In [58], he and others present the NU-Prolog Debugging Environment, which integrates declarative top-down diagnosers for programs with arbitrary body formulas, use of type declarations, static analysis, and test generation capabilities.

Ferrand

In [36] Gerard Ferrand reconstructed Shapiro's framework for Horn clauses, using a declarative semantics tolerating free variables in answer substitutions, and provided results showing some theoretical limitations. These are essentially variations of the following scenario for the wrong clause bug type: a diagnosis is a clause instance, eventually with free variables; therefore there may exist a more specific instance which is correct, with a more specific (and correct) head literal; but if the diagnoser is a logic program returning the clause instance as buggy... then *any* instance of the buggy clause should be buggy.

Since in the present context we're concerned with just getting *a* diagnosis, in particular a diagnosis in terms of the textual program, we don't reach the point where these limitations would be troublesome.

Use of program specifications

In [25][56], W. Drabent and others define a declarative debugging framework using *oracle* (specification) assertions, developing the ideas proposed in [83] and [31]. Their assertions can be simulated in our framework with additional clauses to the oracle statement relation o_s, taking implicit advantage of the subsumed_facet rule, as follows[1]:

• true(G'): o_s(correct,solution(G')).

• false(G'): o_s(incorrect,solution(G')). Notice that G' is a solution, not a goal call.

• posex(G): o_s(incorrect,solution_set∅(G)). G being satisfiable, it must have a solution.

• negex(G): o_s(incorrect,solution(G). Notice that G is a goal call, and this means that any solution for G is incorrect.

In [23] a system is described using *program* rather than oracle specifications, in the form of a correct (although less efficient) version of the buggy program.

Lichtenstein/Shapiro's “Abstract Program Debugging”

In [47], Yossi Lichtenstein and Shapiro defined a framework for abstraction of computation results, to simplify the queries to the oracle, in particular for the debugging of Concurrent Prolog. Their definition of abstraction can also be used for the debugging of programs in higher-level languages, such as object programs executed by a logic program interpreter defining the language.

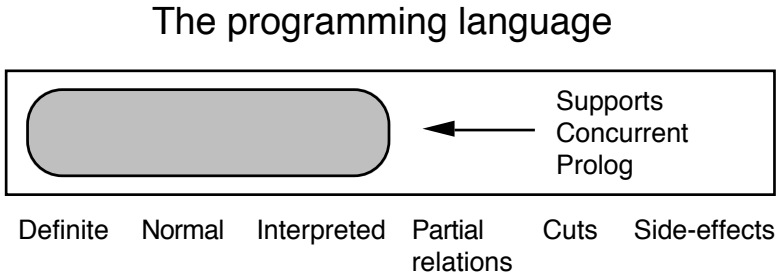

Figure 9.19: Lichtenstein's language territory

Their main contribution, the use of oracle abstractions, is independent of the algorithms used; they use top-down.

---

[1] However we didn't implement this.

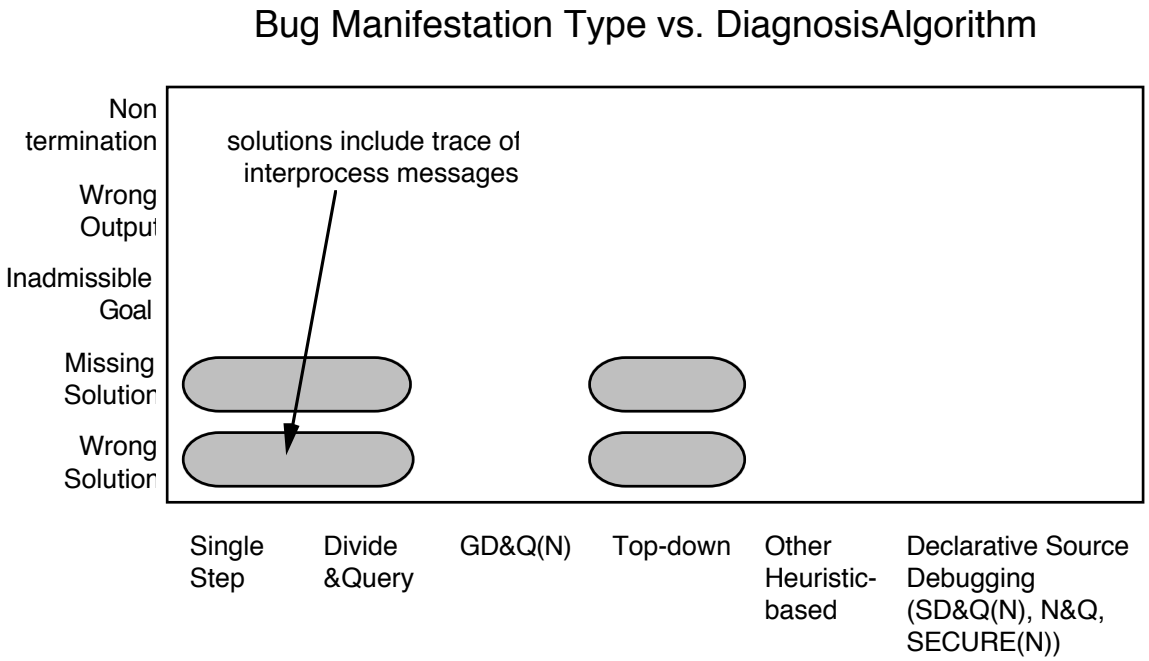

Figure 9.20: Lichtenstein's algorithms

No details on implementation were given.

Huntbach

Although Huntbach's contribution was geared for parallel logic programming languages such as Parlog [40], as a side-effect he came up with a successful approach to deal with Prolog's cuts.

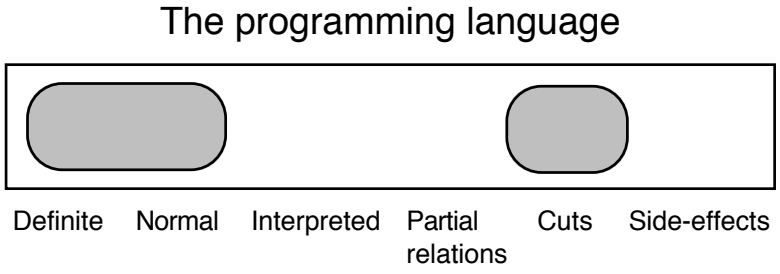

Figure 9.21: Huntbach's language territory

He used the same algorithms as Shapiro:

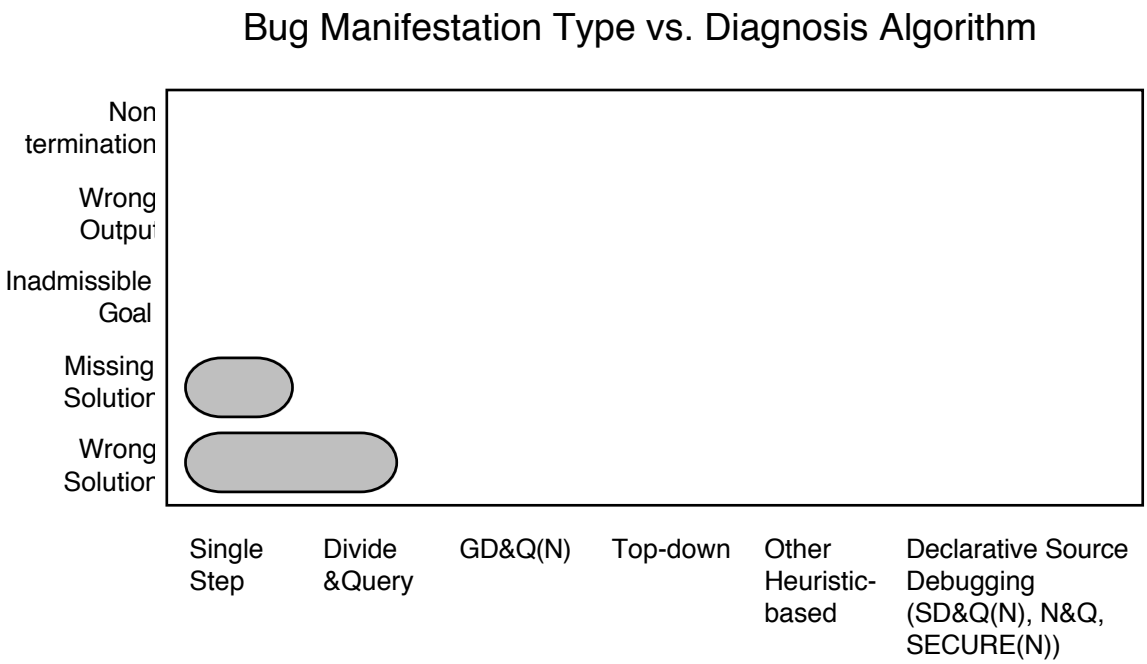

Figure 9.22: Huntbach's algorithms

His implementation stored part of the trace during computation, for the missing solution case, which in the case of a committed-choice language like Parlog is simpler than for Prolog. However it seems also to use recomputation for the wrong solution case.

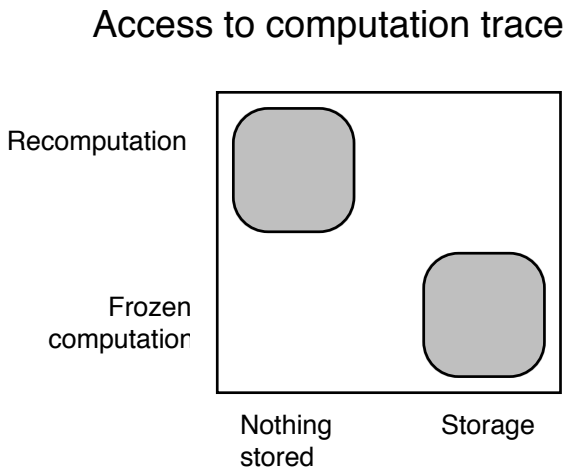

Figure 9.23: Huntbach's implementation

Support for parallel logic programming

Although support for parallel logic programming languages was on our wish list, we weren't able to address it for lack of time. Other people have done it with more or less success, suffering from the non-declarative nature of most of those languages. In all cases the diagnosing algorithms are essencially the same as Shapiro's[83] plus top-down.

Lichtenstein and Shapiro's work on abstraction of computation results [47] (cf. above) was motivated by the problem of declaratively debugging Flat Concurrent Prolog programs [46]. Their basic idea is to associate to each goal solution a trace of inter-process messages relevant for the solution, and to avoid showing it (abstract it) to the oracle if possible (cf. discussion above on pretty-printing, section “Meta-interpreted programs”, chapter 3). Their diagnosis process starts with an “abstract” phase, involving abstract queries, and terminates with one or more “concrete” (non-abstracted) queries.

Mathew Huntbach developed a diagnoser for Parlog[40][41], using the diagnosis algorithms of [83].

Lloyd and Takeuchi developed a diagnoser for Guarded Horn Clauses [51]. They experienced problems with its commit operator, akeen to the problems posed by Prolog's cuts, due to which we had to extend our oracle frameworh (cf. section “Changes to the oracle ”, chapter 4). Later [59] developed a diagnoser using heuristics to choose queries apparently only for the wrong solution problem.

9.1.2.Non-declarative debugging

Lawrence Byrd designed the first Prolog debugger [11], using a 4-port model to describe Prolog's execution, and suporting it through navigation commands. This model persists in most debuggers today, and naturally served us at the implemention level, although the declarative debugger user no longer interacts with it.

Marc Eisenstadt devised a textual tracer [33][32] which among other features included an expert-system like approach to bug detection, through the use of "bug clichés" (and by the way so did [52], in the context of a Prolog tutor). Later Eisenstadt and Mike Brayshaw developed the Transparent Prolog Machine[34], one of the most serious attempts to date to make good use of a graphical user interface for debugging, and incorporating also some existing declarative debugging algorithms [8]. The implementation is based on storing the full computation trace[1].

Mireille Ducassé and Anna-Maria Emde developed a programmable tracer called OPIUM, in connection with the SEPIA environment work at ECRC [26, 27, 28, 29, 30]. Their system can easily be customized in Prolog, using some built-ins to access the tracer features, and thus even declarative diagnosers can be implemented.

Regarding commercial implementations, the only one featuring declarative debugging is BIM Prolog's, which in addition to source tracing [16] supports top-down and divide and query diagnosis as a post-mortem analysis on the computation trace; this is fully stored, albeit in an external database.

There's been more work on logic program debugging, but unrelated to declarative debugging. In addition to commercial tracers, there's for example [77], [90], [19], [24], [60], [86]. A couple of useful surveys of the field are [9] and [28].

9.2. Research problems and opportunities

Following are some issues left unexplored.

[1] Personal communication.

9.2.1. Improving the present framework

Implementations

The first obvious candidate for future work is a more efficient implementation (in terms of space and time usage, not oracle queries), either of the virtual trace storage architecture[1], to achieve the low overhead forecasted in chapter 7, or of another architecture specialized for declarative source debugging; the first declarative source debugging algorithms were just introduced, and better ones may still be designed.

For example, a simplified version of SECURE could be as follows: during execution, and for each program component, keep only the K first goal behavior facets that match it; then apply algorithm SECURE, using only this partial information. This suggests a very efficient implementation, but whose usefulness we're unable to quantify at this point.

A technique which should prove useful in a more efficient implementation is static program analysis, in particular for declarative source debugging. For example, pre-allocating data structures for suspect trees according to the possible goal calls in a program.

Infinite computations were not dealt with in the present framework, because their bug manifestation is eminently non-declarative in nature. But this being one of the most frequent bug types, it seems advisable to have better support in a practical debugger. A possibility is simply to integrate an existing loop checker, to operate on the execution database of an user-interrupted computation.

Now that a Prolog standard is nearer completion (cf. for example [82]), it should be possible to support a substancial part of it, thus maximizing the usefulness of declarative debugging. As a matter of fact this framework already covers most of the proposed Prolog standard requirements, including partial support for the use of I/O streams.

---

[1] Possible with some simplifications regarding the keeping of failures, a major source of overhead; this would have direct consequences on suspect trees for missing solutions, which must be taken into account.

Finally, a production-level implementation would benefit from a good integration with a test-generation system (e.g. [22] [58]). And perhaps the mechanisms for collecting source suspect sets may be reused implementing test generators.

Static program analysis

In addition to providing useful information to make the implementation more efficient, it remains an open question whether static analysis may be used to restrict suspects, or bug types.

For example, given a program that passed a series of LINT-like tests (say, no single variable occurrences in a clause, no undefined predicates, consistent use of types/functors, etc.), can we derive a domain heuristic to affect debugging ? Or could we know when should a source bug manifest more easily through a series of calls, hence affecting the validity of the SECURE assumption?

Oracle

Query abstractions

The use of "pretty-printing" in oracle queries seems essential for applications of declarative debugging to languages built within Prolog. Although the conceptual ground has been determined (cf. section "Meta-interpreted programs", chapter 3, or the abstraction framework of [47]), practical applications beyond Definite Clause Grammars or Concurrent Prolog seem promising.

Potential examples[1]:

- Debugging of Delta-Prolog programs [72][20], by distinctly showing the event trace supporting a goal solution.

- Debugging of Contextual Logic Programs[55], by showing the unit context of a goal call.

- Debugging of Extraposition Definite Clause Grammars[62], showing the extraposition list, and pretty-printing the terminal list.

---

[1] Although some of them may require extensions to the present debugging framework, like adding a goal behavior facet, suspect tree node type and bug type, for example, as was done for output side-effects in chapter 4.

Notion of query difficulty

Oracle queries are not all alike, in the sense that answering them has different "costs" for the oracle[1].

For example, querying about the correctness of goal solution `prime(11)` is probably less costly than querying about the correctness of `noun_phrase( [miguel,hates,examples], [], miguel : X # hates( X, examples) )`. And obtaining answers to the query sequence `{q(a), q(b), xpto(1989)}` seems less stressing to a normal person than `{q(a),xpto(1989),q(b)}`, given the unnecessary "semantic jump" involved in the latter.

Are there general guidelines to establish preference criteria among queries, reliable enough to make its implementation worthwhile ?

This issue has not been dealt with explicitly yet (although the use of abstractions referred in the previous section tries precisely to minimize it). Some debuggers allow either the delaying of oracle answers when it finds them difficult [56], or do not impose a rigid ordering of queries, by letting the user contemplate many at the same time before answering (our own).

9.2.2. Extending the scope of the framework

Obvious candidates are Constraint Logic Programming (CxLP) languages[42], and parallel logic programming languages[84][72], to the community's interest in them on the long term. In both cases the complexity of the control mechanisms typically goes beyond the programmer's ability to visualize them mentally, and thus an automated debugging facility seems appealing.

The main difficulty posed by CxLP is its "non-compositional" nature, in the sense that intermediate goal solutions have associated constraint expressions which typically involve other parts of the derivation. This suggests a hard time for an oracle, answering queries like "Given that <huge constraint expression>, is p(1,X) (such that <another huge expression>) a correct solution?". Query abstractions seem the way to go, but defining good ones seems not trivial.

---

[1] Hence our previous use of the expression "query cost", and its use in the algorithm definitions, although in practice we've taken it to be the number of queries.

Most existing parallel logic programming languages pose two kinds of difficulties: their non-declarative constructs and non-deterministic execution[1]. The first may eventually be dealt with similarly to Prolog cuts, and in fact Huntbach did it for Parlog [40]. The second seems to require either a debugging at the process level, by considering inter-process communication actions as input/output side-effects; or a global approach, involving the storage of a trace global to all involved processes, or execution under a sequencial interpreter.

Another logic programming extension, well-founded semantics [37][65], can be encompassed by our framework's approach[2].

9.2.3. Possible applications outside logic programming

Following is some speculation on the usability of the present framework for other than logic program debugging.

Faultfinding

Our work actually started heading for faultfinding, the idea being of capitalizing on the previous declarative debugging experience to build a generic logic-based faultfinder (as suggested by [64]). Since we found out that enough work was needed in declarative debugging, and since this was more consistent with out institutional context, we sticked to debugging until today. Others have followed the idea with success [39], using Shapiro's original framework.

There is however a crucial difference regarding the use of declarative logic program debugging for faultfinding: the latter requires the generation of tests, and not all suspects are observable, i.e. they can't be queried about.

On the other hand, basic faultfinding principles [79] could find their use in declarative debugging, though this has not been explored.

---

[1] Prolog's sequencial execution being “deterministic”, in this sense, since goal recomputations can be done reproducing exactly the computational behavior - hence our virtual trace storage approach.

[2] Personal communication, Luís Moniz Pereira.

Explanation facilities

Explaining and justifying the result of a computational process to a user is essential, specially when the result is either very important or deriving from complicated mechanisms (e.g. spreadsheet formula[1], expert system deduction). Our suspect sets seem a uniform representation from which to derive explanations in logic program - based systems. Existing expert system explanators usually have difficulties dealing with negative conclusions, and our uniform treatment of missing/wrong solutions seems adequate.

Oracle queries can correspond to intermediate explanations, incorrect oracle statements being requests for explanation, and correct statements meaning that the user has understood the intermediate result ("OK, I understand why sales in Alaska have that value, now continue explaining me (the top bug manifestation, "why we're in the red for next quarter") ")[2]. Therefore diagnosis algorithms become explanation planners, and all the paraphernalia of our framework contributes to give minimal and possibly useful explanations.

Application to other languages

Why logic programming? Could this work be used outside its original scope? Many of the previous chapters dealt with extending the previous declarative debugging frameworks for Prolog, and such improvements may be irrelevant; for example, non-deterministic languages do not require treatment of missing solutions.

However, most of the present approach, just as was the case for [83], can be applied to any language with compositional semantics, i.e. such that its bug manifestations can be characterized by an oracle in a declarative (execution context - free) way, and no causal dependencies exist among intermediate result subcomputations other than through input/output arguments visible to the oracle. Or in our case, we tolerate such dependencies to a certain extent, through our partial treatment of database side-effects.

---

[1] The latest personal computer spreadsheets already come with basic "auditing" features, allowing browsing of antecedent and dependent cell formulas.

[2] Since we really fancy this sort of use for our technology, the HyperTracer diagnosis commands already use the expression "Why this goal behavior facet", instead of a more terse "buggy goal behavior facet".

It just happens that such languages are rare, because programming practice usually makes heavy use of side-effects. Prolog's side-effects are in practice less, and more localized, and hence in general logic programming seems more apt to benefit from the declarative debugging approach. Perhaps the proliferation of object-oriented languages improves the situation, through its stricter encapsulation mechanisms.

Declarative source debugging is one of the contributions of this thesis usable outside logic programming, and seems to mechanize the use of the psychological notion of suspect set human debuggers use (or “slice”, as in [89][1]).

Use in embedded applications

Prolog technology tends to be encapsulated in larger applications. Most software development is usually done in some other language, with a logic program acting as an “inference engine” (cf. for example the repackaging of traditional Prolog systems like Quintus's, in such a way as to allow linking of Prolog code directly with C applications). The ultimate declarative debugger should be able to start debugging C++ or Objective-C code, and transparently continue debugging the encapsulated logic program.

---

[1] He and Lyle explored the idea for the automatic debugging of Fortran and Ada programs in [53]; their “source suspect sets” are the statements affecting a variable with buggy value.

References

1. A.Guessoum and J.W. Lloyd, “Knowledge Base Updates”, Bristol U, May 89.

2. S.P. Abreu, “The ALPES Prolog Programming Environment”, in Proceedings of Workshop on Logic Programming Environments (eds. D. Bowen), Cleveland 1989.

3. J. Alegria, A. Dias and L. Caires, “Towards Distributed Tools for Heterogeneous Logic Programming Environments”, in Proceedings of the 6th International Conference on Logic Programming (eds. Levi and Martelli), MIT Press, 1989.

4. E. Av-Ron, “Top-down Diagnosis of Prolog Programs”, Weizmann Institute, Israel, MSc, 1984.

5. H. Bacha, “Meta Prolog Design and Implementation”, in Proceedings of 5th Int. Conf. on Logic Programming (eds. R. Kowalski and K. Bowen), Seattle, USA, MIT Press 1988.

6. S. Bonnier and J. Maluszynski, “Towards a Clean Amalgamation of Logic Programs with External Procedures”, in Proceedings of Fifth Int'l Conference and Symposium (eds. R. Kowalski and K. Bowen), Seattle, MIT Press 1988.

7. M. Brayshaw and M. Eisenstadt, “Adding Data and Procedure Abstraction to the Transparent Prolog Machine (TPM)”, in Proceedings of 5th Intl. Conf. on Logic Programming (eds. Kowalski and Bowen) Seattle, MIT Press, 1988.

8. M. Brayshaw and M. Eisenstadt, “Declarative Debugging with The Transparent Prolog Machine (TPM)”, in Proceedings of ECAI-88.

9. P. Brna, et al., “An Overview of Prolog Debugging Tools”, DAI, Univ. Edinburgh, Research Paper 398, 1988.

10. M. Bruynooghe and L.M.Pereira, “Deduction revision through intelligent backtracking”, in Implementations of Prolog (eds. J. Campbell), Ellis Horwood, 1984.

11. L. Byrd, "Understanding the control flow of Prolog programs", in Proceedings of the 1980 Logic Programming Workshop (eds. S.-A. Tarnlund), 1980.

12. M. Calejo and L.M. Pereira, "Virtual Trace Storage", Dep.to de Informática, Universidade Nova de Lisboa, Technical Report 1989.

13. M. Calejo and L.M. Pereira, "The HyperTracer Debugging Environment (draft)", CRIA/UNINOVA, February 1990.

14. M. Calejo and L.M. Pereira, "Declarative Source Debugging", in Proceedings of the 6th Portuguese Artificial Intelligence Conference (eds. L.M. Pereira, A. Porto and P. Barahona), Albufeira, Springer Verlag, Lecture Notes on AI, 1991.

15. M. Calejo, L.M. Pereira and A. Porto, "Linguagem Natural por Menus", in Proceedings of 2º Encontro Português de Inteligência Artificial, LNEC, Lisbon, APPIA 1986.

16. A. Callebaut, B. Demoen and R. Venken, "Program Source Oriented Debugging in Prolog", in Proceedings of NACLP'89 Workshop on Logic Programming Environments: The Next Generation (eds. A. Lakhotia, D. Bowen and R. Venken), Cleveland, published by The Center for Advanced Computer Science, Lafayette, LA.

17. W. Clocksin and C. Mellish, "Programming in Prolog", Springer Verlag, 1981.

18. C. Codognet, "Backtracking Intelligent en Programmation Logique - un cadre géneral", U.Paris 7, PhD thesis, 1989.

19. M.J. Coombs, R.T. Hartley and J.G. Stell, "Debugging User Conceptions of Interpretation Processes", in Proceedings of AAAI'86, Morgan-Kaufmann, Science volume, pp. 303-307.

20. J.C.e. Cunha, M.C. Ferreira and L.M. Pereira, "Programming in Delta-Prolog", in Proceedings of the 6th International Conference on Logic Programming (eds. G. Levi and M. Martelli), Logic Programming, MIT Press, Lisbon, 1989.

21. H. Decker, "Deriving Updates from Derivations", June 1989, ECRC, Munich.

22. R. Denney, "Test-Case Generation from Prolog-Based Specifications", in IEEE Software, v. 8, issue 2, 1991.

23. Dershowitz and Y. Lee, "Deductive Debugging", in Proceedings of the Fourth Symposium on Logic Programming, IEEE Computer Society Press, San Francisco, 1987.

24. C. Dichev and B.d. Boulay, "A Data Tracing System for PROLOG Novices", University of Sussex, CSRP 113, 1988.

25. W. Drabent, S. Nadjm-Tehrani and J. Maluszynski, “Algorithmic Debugging with Assertions”, in Meta-programming in Logic Programming (ed. J. Lloyd), MIT Press, Bristol, 1988.

26. M. Ducassé, “Opium+, a Meta-debugger for Prolog”, in Proceedings of ECAI-88, Munich.

27. M. Ducassé, “Scénarios: un paradigme permettant la mise en oeuvre de stratégies de localisation d'erreurs de programmation”, in Proceedings of ERGO-IA, Biarritz, France 1988.

28. M. Ducassé and A.-M. Emde, “State of the Art in Automated Program Debugging”, ECRC TR-LP-25, Munich, Germany, 1987.

29. M. Ducassé and A.-M. Emde, “Automated debugging of real Prolog programs using symptom-driven abstraction. The non-termination analysis.”, ECRC, München, Technical Report, TR-LP-41, July 1989.

30. M. Ducassé and A.-M. Emde, “Opium: a Debugging Environment for Prolog” in Proceedings of NACLP'89 Workshop on Logic Programming Environments: The Next Generation (eds. A. Lakhotia, D. Bowen and R. Venken), Cleveland, published by The Center for Advanced Computer Science, Lafayette, LA.

31. A. Edman and S.-Å. Tärnlund, “Mechanization of an Oracle in a Debugging System” in Proceedings of IJCAI, 1983 .

32. M. Eisenstadt, “Retrospective Zooming, A Knowledge Based Tracing and Debugging Methodology for Logic Programming” in Proceedings of IJCAI, 1985.

33. M. Eisenstadt, “Tracing and debugging Prolog programs by retrospective zooming”, TR-17, HCRL, Open University 1985.

34. M. Eisenstadt and M. Brayshaw, “The Transparent Prolog Machine: an execution model and graphical debugger for logic programming,” Journal of Logic Programming, 5:4, 1988.

35. M.van Emden. “A proposal”, in Logic Programming Newsletter, vol. 1, issue 2, 1981.

36. G. Ferrand, “Error diagnosis in logic programming, an adaptation of E.Y. Shapiro's method”, in Journal of Logic Programming, 4:3, 177-198, 1987.

37. A.V. Gelder, K.A. Ross and J.S. Schlipf, “The well-founded semantics for general logic programs,” Journal of ACM, 221-230, 1990.

38. A. Guessoum and J.W. Lloyd, “Updating Knowledge Bases II”, University of Bristol, Computer Science Department, Technical Report, TR-90-13, May 1990.

39. A. Gupta, "Hardware Diagnosis as Program Debugging" in Proceedings of IJCAI-87 , 524-526.

40. M. Huntbach, "Algorithmic Parlog Debugging" in Proceedings of Fourth Symposium on Logic Programming, San Francisco, IEEE Computer Society Press 1987.

41. M. Huntbach, "Interactive Program Debugging and Synthesis", University of Sussex, PhD, 1990.

42. J. Jaffar and J.L. Lassez, "Constraint Logic Programming" in Procs. POPL 87, 111-119, 1987.

43. T. Kanamori, T. Kawamura and M. Maeji, "Logic Program Diagnosis from Specifications", ICOT, TR-447, March 1989.

44. K.Y. Koh, "Debugging Prolog Programs", Imperial College of Science and Technology, MSC, 1985.

45. Komorowski, "A Declarative Logic Programming Environment," Journal of System Software, 2:8, 1988.

46. Y. Lichtenstein and E. Shapiro, "Concurrent Algorithmic Debugging", in Proceedings of ACM Workshop on Parallel Debugging, Rehovot, published by the Department of Computer Science, The Weizmann Institute of Science 1987.

47. Y. Lichtenstein and E. Shapiro, "Abstract algorithmic debugging" in Procs. 5th International Conference and Symposium on Logic Programming (eds. K. Bowen and R.A. Kowalski), MIT Press, Seattle, 1315--1336, 1988.

48. J. Lloyd, "Declarative Error Diagnosis," New Generation Computing, 5:2, 133-154, 1987.

49. J. Lloyd, "Foundations of Logic Programming", 2nd edition, Springer Verlag, 1987.

50. J. Lloyd and J.C. Shepherdson, "Partial Evaluation in Logic Programming", University of Bristol, UK, Research Report, CS-87-09, December 1987.

51. J. Lloyd and A. Takeuchi, "A framework for debugging GHC", TR-186, ICOT, Tokyo, Japan, 1986

52. C.-k. Looi and P. Ross, "Debugging Prolog Programs in an Intelligent Tutoring System", DAI, Univ. Edinburgh, DAI Research Paper No. 308, May 1987.

53. J.R. Lyle and M. Weiser, "Automatic Program Bug Location by Program Slicing" in Proceedings of The Second International Conference on Computers and Applications, Peking, China 1987.

54. H. Manilla and E. Ukkonen, “Timestamped term representation for representing Prolog”, in Proceedings of Symposium on Logic Programming (eds. I. Press), 159-165, 1986.

55. L. Monteiro and A. Porto, “Contextual Logic Programming”, in Proceedings of the 6th International Conference on Logic Programming (eds. G. Levi and M. Martelli), Logic Programming, MIT Press, Lisbon, 284-299, 1989.

56. S. Nadjm-Tehrani, “Contributions to the Declarative Approach to Debugging Prolog Programs”, Linköping Univ., Licenciature thesis, 1989.

57. L. Naish, “Declarative Diagnosis of Missing Answers,” in New Generation Computing Journal, 1990

58. L. Naish, P.W. Dart and J. Zobel, “The NU-Prolog Debugging Environment”, in Logic Programming - Procs. of the 6th Intl. Conf. (eds. G. Levi and M. Martelli), MIT Press, Lisbon, 1989.

59. E. Nicholson, “Declarative debugging of the parallel logic programming language GHC”, in Proceedings of the Eleventh Australian Computer Science Conference, Brisbane, Australia, 1988.

60. M. Numao and T. Fujisaki, “Visual Debugger for Prolog”, in Procs. of the Second Conference on Artificial Intelligence Applications, IEEE Computer Society Press, 1985.

61. F. Pereira and et. al., “C-Prolog User's Manual”, Dept. of Architecture, University of Edinburgh, UK, 1973.

62. F. Pereira and Shieber, “Prolog and Natural-Language Analysis”, Center for the Study of Language and Information, Stanford, 1987.

63. L.M. Pereira, “Rational debugging of logic programs”, in Proceedings of the 1st Portuguese AI Meeting , Associação Portuguesa Para a Inteligência Artificial, 1985.

64. L.M. Pereira, “Rational debugging in Logic Programming”, in Procs. Third International Conference in Logic Progamming (ed. E. Shapiro), Lecture Notes in Computer Science, Springer-Verlag, 1986.

65. L.M. Pereira, J.N. Aparício and J.J. Alferes, “Nonmonotonic Reasoning with Well Founded Semantics”, in Proceedings of Eighth International Conference on Logic Programming (ed. K. Furukawa), Paris, MIT Press, 475-489.

66. L.M. Pereira and M. Calejo, “Debugging Errors in Logic Programs”, in Proceedings of the 3rd Portuguese AI meeting, Associação Portuguesa Para a Inteligência Artificial, Braga, 1987.

67. L.M. Pereira and M. Calejo, “Debugging: motivation, framework and specification”, DI/FCT/UNL, ALPES/ESPRIT P-973 report, 1987.

68. L.M. Pereira and M. Calejo, "A framework for Prolog debugging", in Proceedings of 5th Intl. Conf. on Logic Programming (eds. Kowalski and Bowen), Seattle, MIT Press, 1988.

69. L.M. Pereira and M. Calejo, "Algorithmic Debugging of Prolog Side-Effects", in Procs. EPIA 89 (eds. J.P. Martins and E. Morgado), Lecture Notes in Artificial Intelligence, Springer-Verlag, Lisbon, 1989.

70. L.M. Pereira, M. Calejo and J.N. Aparício, "Refining Knowledge Base Updates", UNINOVA, AI Center, COMPULOG report, November 1989.

71. L.M. Pereira, M. Calejo and J.N. Aparício, "Refining Knowledge Base Updates" in Proceedings of Simpósio Brasileiro de Inteligência Artificial 1990 .

72. L.M. Pereira, L. Monteiro, J. Cunha and J.N. Aparicio, "Delta Prolog: a distributed backtracking extension with events", in Procs. Third International Conference on Logic Programming (ed. E. Shapiro), London, UK, 1986.

73. L.M. Pereira, F. Pereira and D. Warren, "User's Guide to DECsystem-10 Prolog", Laboratório Nacional de Engenharia Civil, Lisboa 1978.

74. L.M. Pereira and A. Porto, "Intelligent backtracking and sidetracking in logic programs - the theory", CIUNL report 12/79, Universidade Nova de Lisboa, 1979.

75. L.M. Pereira and A. Porto, "Selective Backtracking", in "Logic Programming" (eds. Clark and Tärnlund), Academic Press, 1982.

76. D.A. Plaistead, "An Efficient Bug Location Algorithm", in Proceedings of 2nd. Int'l Conf. on Logic Programming, Uppsala, Sweden, 1984.

77. D. Plummer, "Coda: An Extended Debugger for PROLOG", in Proceedings of 5th Int. Conf. on Logic Programming (eds. Kowalski and Bowen), Seattle, MIT Press 1988.

78. A. Porto, L.M. Pereira and L. Trindade, "UNL Prolog Reference Manual", Universidade Nova de Lisboa, ALPES ESPRIT P-973 Report, 1987.

79. R. Reiter, "A Theory of Diagnosis from First Principles," in Artificial Intelligence Journal, 32, 57-95, 1987.

80. P. Roussel, "Prolog: Manuel de reférence et d'utilization", Groupe d'Intelligence Artificielle, Marseille-Luminy, 1973.

81. D. Sahlin, "An Automated Partial Evaluator for Full Prolog", Swedish Institute of Computer Science, PhD, 1991.

82. R.S. Scowen, "PROLOG - Draft for Working Draft 5.0", International Organization for Standardization, Input for Working Draft, ISO/IEC JTC1 SC22 WG17 N72, June 1991.

83. E. Shapiro, “Algorithmic Program Debugging”, MIT Press, 1983.

84. E. Shapiro, “The Family of Concurrent Logic Programming Languages,” ACM Computing Surveys, 21:3, 412-510, 1989.

85. R. Stallman and G.J. Sussman, “Forward Reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis,” Artificial Intelligence Journal, 9, 135-196, 1977.

86. H. Takahashi and E. Shibayama, “PRESET - A Debugging Environment for Prolog”, in Logic Programming'85 (ed. Wada), Springer-Verlag 1985.

87. A. Tomasic, “View Update Translation via Deduction and Annotation”, in Procs. ICDT'88 , Springer-Verlag 1988.

88. D.H.D. Warren, “The Andorra Model”, in Proceedings of Gigalips Project Workshop, Manchester 1988.

89. M. Weiser, “Programmers Use Slices When Debugging,” in Communications of the ACM, 25:7, 446-452, 1982.

90. J.F.H. Winkler, A.van Reeken and A. Schleiermacher, “A Prolog Debugger Based on a Refined Box Model”, in Proceedings of Logic Programming Environments ICLP'90 Preconference Workshop (eds. M. Ducassé, A.-M. Emde, T. Kusalik and J. Levy), Eilat, Israel, published by European Computer-Industry Research Center, Munich.

Index