1 Project history

The official project start date was October 1994. Its foreseen duration 18 months. The time slippage requires some explaining.

During the first part of 1995 we negotiated the specification and acquisition of the equipment from DEC Portugal. The actual installation of the Alpha machines was also delayed during 1995 because our department moved wholesale to a different building.

After the machines were individually installed by us, we still didn't have the FDDI-based interconnection hardware. Due to a series of difficulties, DEC Portugal was only able to complete the installation of the FDDI hardware ring by February 1997. After their installation, it took us a few months to complete the full software installation, so the experimentation on the cluster configuration started only after April 1997.

Since then the following steps had to be performed in order to fulfill the project goals:

1. To install our NanoProlog system in the Alpha processors.
2. To install our PVM-Prolog system, based on NanoProlog, in the Alpha processors.
3. To start the experimentation with newly parallelized versions of Prolog interpreters and the diagnosis prototypes.

The third step began only in September 1997, corresponding to our sending a revised schedule and our plan to complete the whole project by the end of 1997.
Because of unforeseen personal problems of two researchers, we could not complete the revised schedule by the end of 1997. Then the project leader was away on sabbatical in California during the whole of January and February 1998. Upon his return, beginning of March, the coordination of the writing up of this final report was resumed.

We warmly thank DEC of their support of this project, and remain available for any clarifications on our work.
2 Project Development

This research project targeted the evolution of Prolog towards a more expressive logic programming language, tied to a distributed execution environment exploiting innovative implementation techniques, and its application to diagnosis. Next, the language development is described first, followed by the distributed environment support, followed by the applications part. An appendix contains the abstracts of our publications for easier reference.

2.1 Language Development

Task Coordinator: Luis Moniz Pereira

Participants: J. J. Alferes, C. Damásio

At the theoretical level, before the project started we had developed a logic programming language with explicit negation, its declarative semantics, and contradiction removal mechanism for the language. We’ve shown the expressivity and usefulness of the language (see our book [AP96] and references therein). The main idea regarding the present EERP project was to take this language, and to develop procedures and system support to execute it in a distributed environment, as well as to test it all on diagnosis applications.

With a view to this, we started by developing top-down query evaluation procedures for logic programs with explicit negation that (for allowed programs) avoids the problems with looping (according to the well-founded semantics). This work is reported in [ADP94b, ADP94a, ADP95]. The procedures were first implemented by means of a meta-interpreter in a (non-distributed) Prolog.

By adding to the logic programming language such an explicit form of declaring falsity, programs become liable to contradiction. When a program is contradictory its revision is in order. So, we developed contradiction removal methods, and a procedure (REVISE) for revising contradictory programs with integrity constraints [DNP94] in order to render them consistent. We implemented the procedure in Prolog (the REVISE system is stored by us and usable on examples at the international software library: http://www.uni-koblenz.de/ag-ki/LP/system.REVISE.html).

Finding a diagnosis for a faulty behaviour of a system, the main issue addressed in the area of model based diagnosis, is a problem for which our new language of logic programs with explicit negation and contradiction removal is well suited. If the system’s behaviour and the observed faulty behaviour are described by a (contradictory) logic program, removing the contradiction is tantamount to finding the diagnosis of the system, by identifying a mode of faulty behaviour consistent with the observations as well as explaining them. So, we used our language and its contradiction removal procedure to implement diagnosis problems. We then studied how to implement preferences among various diagnosis, and strategies in diagnosing in [DNP95]. We next tested our approach in various problems of diagnosis of distributed systems (see section 2.3
below for more details on this work. The distribution is achieved by relying on our PVM-Prolog, which was further developed within this project (cf. next section).

However, the advent of efficiently implemented tabulation mechanisms in query answering procedures for logic programs, not available at the time the project was conceived and proposed, made us adopt a radically new course regarding the interpreters and procedures for the extended logic programming language. Tabulation not only provides new soundness and completeness results for logic programs with loops and variables, but also greater efficiency, including for distributed computations.

The XSB-Prolog system, developed by the XSB-Group of the Computer Science Department of SUNY (State University of New York) at Stony Brook, is based on SLG resolution [CW93], which uses a form of tabling in addition to SLD resolution. SLG evaluates definite programs by keeping tables of sub-goals and their associated answers, and by resolving repeated instances of subgoals against answer clauses from the tables, rather than against program clauses. This way, it avoids looping. In practice, not all subgoals need to be tabled for looping to be avoided. XSB-Prolog allows for a declaration of the tabled predicates. For these SLG resolution is used, whilst for non-tabled ones SLD resolution is used instead. If no predicates are tabled XSB-Prolog behaves just as an efficient Prolog.

For programs with default negation, when a negative ground\(^1\) goal not \(A\) is called (in XSB-Prolog notation, \(\texttt{not}(A)\)). \(A\) has to be completely evaluated before the success of \(\texttt{not}(A)\) can be determined. In [CW93] an algorithm for detecting tabled calls that are completely evaluated is defined. It is shown there that the algorithm thus specified computes answers under the well-founded semantics [GRS91] of normal logic programs.

With the availability of tabling, instead of the emphasis being put on distributing independent subgoals, which are normally rare to encounter, the emphasis becomes reusing computation results via tabling, in a first step, and via distributed tabling in a second step. Also, our attempts to find good distributed diagnosis applications thwarted our expectations. Instead, we realized that it was best to distribute the revision algorithm itself used in diagnosis. But whereas XSB-Prolog provides for efficient non-distributed tabulation, which we immediately made use of, for the second step we had to develop our own distributed tabulation theory and algorithms. In fact this is still ongoing and the present implementation is written yet at a high level, in PVM-Prolog.

With regard to the first step, based on the theoretical results of [DP95] we implemented a preprocessor for programs in our extended language into the language of normal programs of XSB-Prolog\(^2\). We ported the revision implementation (the REVISE system) to XSB-Prolog. We also ported into XSB-Prolog our PVM-Prolog facilities. Finally, XSB-Prolog itself had to be ported to the

---

\(^1\)Negative non-ground calls are delayed until its arguments are ground. If in the end that is not possible, those calls flounder.

\(^2\)For more information on XSB-Prolog, and to obtain the software, see http://www.cs.sunysb.edu/~sbprolog/xsb-page.html
64-bit architecture of the Alpha machines. All this work was carried out in collaboration with the XSB-group of SUNY at Stony Brook via the joint 3-year project REAP, funded by the American NSF and the Portuguese FLAD (the Portuguese-American Foundation for Development).

We next addressed the second step. Indeed, even together with the tabulation mechanism of XSB-Prolog, the REVISE procedure implementation is not suited for the explicit distribution of goals. In fact, goals are usually dependent on one another, and the requirement of minimality in the removal of contradiction demands a global analysis of the program. To take advantage of parallelism, and in order to obtain greater efficiency in computing diagnoses, we started to develop, from scratch, a distributed tabulation method (based on the results of the Lisbon PhD thesis [Dan96]) and, based on a suspend/resume mechanism, to implement a distributed revision system. This revision system plus distributed tabulation was implemented using the PVM-Prolog developed in this project, and tested with some of our application examples in the domain of model based diagnosis. The system, described in the report [DA98], includes a top level query mechanism for (cyclic) definite logic programs, and a revision procedure for normal logic programs with integrity constraints. Query to normal programs can now be made via the revision procedure in a simple way. However, for efficiency, we intend to improve the system such that queries to normal programs can be handled directly, without the need for the revision procedure. Debugging tools have been developed for this system, as described in the aforementioned report.

The new system does not yet handle logic programs with explicit negation, as foreseen in the proposal. However, for application in diagnosis, the use of explicit negation can be replaced by the use of integrity constraints, which is already supported by the system.
2.2 Distributed Environment Support

Task Coordinator: José C. Cunha

Participants: Rui Marques, Vítor Duarte, Pedro Medeiros, João Lourenço, Joaquim Gonçalves

A high degree of flexibility and portability was required to support the experimentation conducted in this project. Such requirement was met by defining the PVM-Prolog in the following way:

1. An intermediate programming layer was defined based on a set of built-in Prolog predicates supporting parallelism and distribution control, and interprocess communication. Specific strategies for parallel and distributed execution, and communication, have been implemented on top of such Prolog-based interface with reduced development effort.

2. By using PVM as the runtime platform, a high degree of portability was automatically achieved, as well as support for heterogeneity at the architecture and operating system levels.

2.2.1 The PVM-Prolog Model

PVM-Prolog [MC95a, MC95b, CM96, CM97] consists of two orthogonal components:

1. A Prolog interface to PVM provides predicates for the spawning of independent Prolog evaluators (as PVM tasks), in local and remote processors of a PVM virtual machine, including the passing of top goals to such Prolog tasks. PVM-like communication predicates are provided so that Prolog terms can be asynchronously exchanged by Prolog tasks. Complete control of a Prolog task is possible by using this interface.

2. A multi-threaded Prolog model supports internal concurrency to a PVM-Prolog task, in the form of multiple threads. Each thread solves an individual goal within the scope of a PVM-Prolog task. Predicates for thread creation, destruction and control are provided. Within a task, communication between threads is supported by shared term queues with put and get predicates which are implemented in the address space of the enclosing task. Threads in distinct tasks can directly communicate using the above PVM interface. Alternatively, incoming PVM messages and signals can be bound to and automatically saved in term queues within each destination task for later collection by local threads.

The PVM-Prolog interface has provided adequate support to a very effective and rapid prototype development for the already described application tasks in this project. Additional flexibility has been achieved by providing a lower level set of Prolog predicates (defining what we call the PVM-Prolog0 interface) that allow access to all basic PVM functions from a Prolog program, namely:
1. The spawning of PVM tasks which can execute Prolog or any kind of executable files, i.e. C programs. This allows PVM-Prolog to be used for the implementation of generic control shells.

2. The communication between PVM-Prolog tasks and C/PVM tasks, by using built-in predicates for the packing, conversion and passing of Prolog terms and C structures. This can be used to compose heterogeneous multi-lingual applications.

The multi-threaded model allows a clearer implementation of reactive agents, i.e. that can handle external events (namely, signals and messages from other agents) by asynchronously activating user-defined predicates. Without threads, one is typically forced to specify a kind of eternal process with a service loop for event handling, leading to more complex and error-prone code. In PVM-Prolog we are able to program such kind of behavior entirely in Prolog, by relying on our multi-threaded component.

Threads also allow each agent to perform internal goal evaluation strategies which can exploit concurrency, or even parallelism if a shared-memory multiprocessor is available. This latter possibility was not exploited in this project but it is an interesting issue.

Experimentation with the PVM-Prolog is reported in the sections above. Further experimentation is described in [SMWC97] where PVM-Prolog is used to support the implementation of an agent-based language.

2.2.2 PVM-Prolog Implementation

The following implementations have been developed using distinct Prolog engines:

1. Based on NanoProlog [Dia90]. This supports both the PVM interface and the multi-threaded component. Threads are implemented at user-level, by modification of the NanoProlog abstract machine.

2. Based on SICSTUS-Prolog [CW88]. This only supports the PVM-based interface.

3. Based on XSB-Prolog [SW94]. This only supports the PVM-based interface.

2.2.3 XSB-Prolog port to the 64-bit Alpha architecture

XSB-Prolog being the only available Prolog system that implements tabling, was designed for 32 bit machines and not able to run using a 64 bit address space. In order to take full advantage of the 64 bit Alpha architecture some changes had to be made concerning several past implementation decisions which implied a 32 bit word machine. The new version is fully compatible with the old (32 bit) one.
2.2.4 Monitoring PVM-Prolog Programs

In order to support the monitoring of PVM-Prolog programs, several experiments were performed. The goal was to evaluate existing PVM monitoring functionalities in order to make them accessible to PVM-Prolog programs, and whenever necessary to develop new tools with the following requirements:

1. To be able to trace and monitor user-defined Prolog predicates.
2. To be able to trace and monitor PVM-related predicates, namely concerning process spawning and communication.
3. To be able to access and visualize the trace information by using existing off-line and on-line visualization packages.

The results of our experimentation are described in [CD98, CDLM98].

2.2.5 Further System-level Experimentation with the Alpha Processors

We have developed a model and infrastructure for heterogeneous distributed computing applications. The PHIS model [MC97a, MC97b] currently supports the interconnection of heterogeneous applications built from PVM and MPI components, and it runs on a UNIX local network with distinct nodes including UNIX workstations, the Alpha farm, and a Transputer-based multicomputer (a Meiko CS-1 with 16 transputer nodes).

Concerning parallel applications, we are developing an integrated environment that supports the parallel and distributed execution of genetic algorithms using the PHIS model as a basis for the interconnection between distinct components [CML98].

2.2.6 Ongoing Work

Ongoing work includes the development of a distributed debugging system for C/PVM and C/MPI programs that will run on the above mentioned heterogeneous platform [CLA96, LCK97, LC98b, CLD98, CLA98, KCD97, CLV98, LC98a]. Such debugging system is being used to implement a distributed debugger for PVM-Prolog.
2.3 Application to Diagnosis

Task Coordinator: Luís Moniz Pereira

Participants: J. J. Alferes, C. Damásio, I. Móra, M. Schroeder

We have studied the domain of model based diagnosis using logic programming. This work was carried out in collaboration with Prof. Wolfgang Nejdl's group in the University of Hannover via a German-Portuguese INIDA project (mentioned in the PADIPO project proposal), which supported several missions of German researchers to Lisbon and of researchers from our group to Hannover.

In [DNPS95] we studied the representation of preferences and strategies of diagnosis by the use of logic programming. This work was further detailed in the Hannover PhD thesis [Sch97] mostly carried out in Lisbon (an improved version is being published by Kluwer), and where many (not necessarily distributed) diagnosis examples can be found.

Regarding implementation of distributed diagnosis systems using the PVM-Prolog and/or the interpreters developed in the project, we've mainly tackled two domains: distributed diagnosis of computer networks, and diagnosis of communication protocols.

Distributed Diagnosis of a Computer Network In the implementation we've considered a computer network with four machines. To achieve fault-tolerant diagnosis each machine is seen as a diagnosis agent or as a tester agent. In our case, three agents \(a, b, c\) diagnose another agent \(d\). Since \(a, b, c\) have the same description of \(d\) their findings are redundant in case they are up and they receive the same test result of \(d\). The tasks of the agents can be divided into four steps. Initially, the agents receive a start message from the creator process. In a second step, the diagnosis agents send requests for the results to the group of agents whose underlying hardware is examined, the tests are executed and the requests are answered. Thirdly, the diagnosing agents compute diagnoses based on the new test results they received and communicate their diagnoses among the diagnosis group. Finally, each diagnosis agent computes a consensus based on all diagnoses it received, and sends it to the (possibly human) agent it has to report to. The work is detailed in [dAMA95, SdAMP96, SW99].

Diagnosis of a Communication Protocol Here we considered a distributed system with \(n\) nodes, e.g. a computer network consisting of \(n\) machines. An agent-based approach decomposes a system into a set of subsystems. Each subsystem is diagnosed by an agent which has detailed knowledge over its subsystem and an abstract view of the neighbor subsystems. Most failures can be diagnosed locally within one subsystem. In the case of the computer network most machines in a subnet can usually fail without affecting machines in other subnets. Only those computers in other subnets can be affected which have sent to the faulty machine. Moreover, the local computation of diagnoses
avoids the communication overhead which would be needed to forward all observations to the central diagnosis engine. Failures which affect more than one subsystem are diagnosed by the agents cooperating with each other. The cooperation process is triggered locally by an agent, when it realizes that it cannot explain the observations by a failure in its own subsystem. The work is detailed in [FdAMNS97, SW98].

2.4 Conclusions

We have provided the following software, which we can demonstrate:

- A PVM-Prolog interface in the form of 'C' built-ins. We ported it to our NANO-Prolog with threads. It was also ported to SICSTUS-Prolog and XSB-Prolog (32-bit version).
- A XSB-Prolog port to the 64-bit Alpha architecture. This was also extended with the PVM-Prolog interface.
- An interpreter in Prolog for executing SLX, a top-down proof procedure for an extended logic programming language and semantics (WFSX).
- A preprocessor of extended logic programs into normal logic programs for XSB-Prolog to make use of its tabulation facilities.
- The REVISE interpreter, in Prolog, for performing contradiction removal, which was also ported to XSB-Prolog.
- Application code to model distributed and non-distributed diagnosis problems.

We have provided separately a number of theses, publications, and reports, describing in detail the work done, whose references and abstracts appear below.

References


A Abstracts of the papers

A.1 Language Development

[ADP94b] “Top-down query evaluation for well-founded semantics with explicit negation” In this paper we define a sound and complete top
down semantic tree characterization, that includes pruning rules, of the well
founded semantics for programs extended with explicit negation (WFSX), and
compare it to other related approaches. It is amenable to a simple implemen-
tation, and by its nature readily allows pre-processing into Prolog, showing
promise as an efficient basis for further development.

[ADP94a] “SLX - A top-down derivation procedure for programs
with explicit negation” In this paper we define a sound and (theoreti-
cally) complete top-down derivation procedure for a well-founded semantics of
logic programs extended with explicit negation (WFSX). By its nature, it is
amenable to a simple interpreter implementation in Prolog, and readily al-
 lows pre-processing into Prolog, showing promise as an efficient basis for further
development.

The derivation method is directly inspired on the semantic AND-tree char-
acterization of WFSX in our paper [ADP94b], but does not require knowledge
of it. In fact, that work can be seen as a preliminary step towards the definition
of the procedure presented here.

Because of its properties, which are paramount for top-down query evalua-
tion and other approaches do not fully enjoy, WFSX is a natural candidate to
being the semantics of choice for logic programs extended with explicit negation.
Moreover, WFSX is sound wrt to the answer-sets semantics, and it is a better
approximation to answer-sets than simply using the well-founded semantics of
normal programs, plus a renaming of explicitly negated literals. Thus, our top-
down procedure can be used as a sound one for answer-sets, that provides less
incompleteness than others.

Since WFSX coincides with the well-founded semantics on normal pro-
grams, our method is applicable to it and, for ground programs, compares fa-
vorably with previous approaches [ADP94b].

[ADP95] “A Logic Programming System for Non-monotonic Reason-
ing” The evolution of Logic Programming semantics has included the intro-
duction of a new explicit form of negation, beside the older implicit (or default)
notation typical of Logic Programming. The richer language has been shown
adequate for a space of knowledge representation and reasoning forms.

The widespread use of such extended programs requires the definition of
a correct top-down querying mechanism, much as for Prolog wrt. normal pro-
grams. One purpose of this paper is to present and exploit a SLDNF-like deri-
vation procedure, SLX, for programs with explicit negation under well founded
semantics (WFSX) and prove its soundness and completeness. (Its soundness
wrt. the answer-sets semantics is also shown.) Our choice of WFSX as the base
semantics is justified by the structural properties it enjoys, which are paramount for top-down query evaluation.

Of course, introducing explicit negation requires dealing with contradiction. Consequently, we allow for contradiction to appear, and show moreover how it can be removed by freely changing the truth-values of some subset of a set of predefined revisable literals. To achieve this, we have in this paper introduced a paraconsistent version of \( WFSX \), \( WFSX_\beta \) that allows contradictions, and for which our \( SLX \) top-down procedure is proven correct as well.

This procedure can be used to detect the existence of pairs of complementary literals in \( WFSX_\beta \) simply by detecting the violation of integrity rules \( \bot \leftarrow L, \neg L \) introduced for each \( L \) in the language of the program. Furthermore, integrity constraints of a more general form are allowed, whose violation can likewise be detected by \( SLX \).

Removal of contradiction or integrity violation is accomplished by a variant of the \( SLX \) procedure which collects, in a formula, the alternative combinations of revisable literals' truth-values that ensure the said removal. The formulas, after simplification, can then be satisfied by a number of truth-values changes in the revisables, among "true", "false", and "undefined". A notion of minimal change is defined as well that establishes a closeness relation between a program and its revisions. Forthwith, the changes can be enforced by introducing or deleting program rules for the revisable literals.

To illustrate the usefulness and originality of our framework we applied it to obtain a novel logic programming approach, and results, in declarative debugging and model based diagnosis problems.

[DNP94] "REVISE: An Extended Logic Programming System for Revising Knowledge Bases" In this paper we describe REVISE, an extended logic programming system for revising knowledge bases. REVISE is based on logic programming with explicit negation, plus a two-valued assumption revision to face contradiction, encompasses the notion of preference levels. Its reliance on logic programming allows efficient computation and declarativity, whilst its use of explicit negation, revision and preference levels enables modeling of a variety of problems including default reasoning, belief revision and model-based reasoning. It has been implemented as a Prolog meta interpreter and tested on a spate of examples, namely the representation of diagnosis strategies in model-based reasoning systems.

[DNPS95] "Model-Based Diagnosis Preferences and Strategies Representation with Meta Logic Programming" Preferences and strategies are fundamental to model-based diagnosis, for specifying preferred and fall-back approaches to the diagnosis task, both to capture general and domain specific criteria, but also to tackle the complexity issue by employing heuristics. A formal framework based on extended logic programming and meta-programs is provided to represent preferences and strategies required by model-based diagnosis. This framework is clearer and more expressive than other approaches that
have addressed these problems. We show how the concepts of preferences and strategies are directly programmed and captured by logic meta-programming and meta-reasoning methods, and their implementation techniques.

The paper is intended as proof-of-principle that all concepts needed by a model-based diagnosis system can be represented declaratively and captured by a logic meta-program. Specialized more efficient algorithms can be substituted for the simpler proof-of-principle ones we include, and are the subject of ongoing work.

[DP95] "Abduction over 3-valued extended logic programs" One major contribution of this paper is that of tackling contradiction, with some generality, and in particular within our choice semantics of extended logic programs, WFSX. Although such issues can be approached from a belief revision point of view, they can likewise, and equivalently, be thought of within an abductive framework. Indeed, a significant result of this paper is to define such three-valued abductive framework, directly over three-valued extended logic programming semantics.

We allow for contradiction to appear and show moreover how it can be removed by freely changing the truth-values (among "true", "false", and "undefined") of a subset of pre-designated revisable literals: the abducibles. To do so, we start by introducing a paraconsistent version of WFSX, WFSXp, that permits reasoning with contradictory programs. Consistency is regained by revising the truth value of the revisables that lend "support" to integrity constraint-violating literals, which are detected in WFSXp.

A notion of global minimal revision (improving on previous work in the literature) is defined as well, which establishes a closeness relation between the several abductive solutions, thereby defining minimality of abduction too. Forthwith, the changes required to achieve program revision can be enforced by introducing or deleting program rules, for the abducible literals only, without loss in generality.

Subsequently, we show how other abductive frameworks relate to our own (namely Generalized Stable Models). We also demonstrate our framework can be used to characterize various logic programming semantics, namely Stable Models, Stationary Models, Well-Founded Semantics, Preferred extensions, Extended Partial Stable Models and Answer Sets.

A final result of the paper is that of proving that Generalized Stable Models and the abductive framework defined here have the same expressive power.

A.2 Distributed Environment Support

[SMWC97] CAP – Concurrent Action and Planning: Using PVM-Prolog to Implement Vivid Agents "First, we describe PVM-Prolog, a Prolog core extended by an interface to PVM, the Parallel Virtual Machine, a standard software which allows to view a network of heterogeneous machines as a single parallel computer. Besides PVM's coarse-grain parallelism, PVM-Prolog
includes a process-internal thread concept to realize fine-grain concurrency. Second, we review the concept of vivid agents and develop the architecture CAP (Concurrent Action and Planning) that serves as an operational semantics for vivid agents. Finally, we merge the above two lines of research by showing that PVM-Prolog is an excellent candidate to implement vivid agents and multi-agent systems. In general: the coarse-grain parallelism is used to spawn agents in a network, while the fine-grain concurrency is used to run a perception-reaction cycle and a planning facility for each agent concurrently.

[CM97] Distributed Algorithm Development with PVM-Prolog "The design of parallel and distributed algorithms poses many difficulties. This has motivated proposals of high-level languages for parallel program composition, their physical distribution in multiprocessor architectures, specification of communication and synchronization, and failure handling. In this paper, we describe an approach for prototyping parallel and distributed programs, that is based on a logic programming system called PVM-Prolog. PVM-Prolog is a programming interface from Prolog to the PVM system, offering all PVM functionalities to the logic programmer such as process spawning and control, virtual machine management, and failure handling. We describe the PVM-Prolog model and illustrate its application."

[MC95a] PVM-Prolog: Parallel Logic Programming in the PVM System "In this document we propose a parallel programming model which encompasses the functionalities of a multi-threaded Prolog and the ones of a parallel programming interface from Prolog to the PVM system.

While the multi-threaded model was designed anew, the Prolog/PVM interface basically corresponds to the inclusion of additional system predicates which give full access to the PVM environment, from a Prolog program.

This document briefly introduces the basic concepts of the proposed model, and then describes in detail all the supported extensions and additional system predicates."

[MC95b] Using PVM with a Logic Programming Interface "In this document we summarize the functionality of a Parallel Logic Programming system based on PVM. The Prolog interactive interpreter, with its powerful built-in mechanisms of unification and backtracking is particularly well suited for dynamic interaction with the parallel system.

Although Logic Programming has a declarative nature, Prolog has also a procedural interpretation upon which we rely for direct mapping of PVM routines. To conclude we give some ongoing and future work involving our current PVM-Prolog prototype and its use for the development of applications."

[CM96] PVM-Prolog: A Prolog Interface to PVM "PVM-Prolog is a programming interface from Prolog to the PVM system. Parallel and distributed artificial intelligence applications, such as multi-agent systems and distributed
knowledge and database systems, can directly be programmed using this interface. Multiple distributed Prolog processes can cooperate using a message-passing model and can have access to all the functionalities provided by the PVM environment, such as process spawning and control, virtual machine management, and failure handling. The model can also be used for rapid prototyping of parallel program schemes under PVM, high-level management and control of the PVM environment, implementation of parallel and distributed logic programming languages, hybrid or multi-lingual distributed programming systems consisting of PVM-C, PVM-FORTRAN, and PVM-Prolog components, and for the support of heterogeneous logic computing environments exploiting forms of explicit and implicit parallelism."

[CaLa96] A Debugging Engine for a Parallel and Distributed Environment "This paper describes a debugging interface that has been developed for a parallel software engineering environment and that was developed on top of the PVM environment in the scope of the SEPP and HPCTI projects of the COPERNICUS Programme. The main goal of this interface is to provide the basic debugging functionalities that are required by some components of that environment. We give special attention to the requirements posed by high-level tools of the environment, and to the need of providing a flexible debugging support layer that can be suitably adapted and extended. We present the system logical architecture and the interface specification of the debugging engine. We discuss its interfacing with other components of the environment, namely a graphical editor for the GRAPNEL visual parallel programming language, and a testing tool. We finally describe current work on the improvement of the debugging engine."

[CLA96] A Distributed Debugging Tool for a Parallel Software Engineering Environment "We discuss issues in the design and implementation of a flexible debugging tool and its integration into a parallel software engineering environment."

[LCK+97] An Integrated Testing and Debugging Environment for Parallel and Distributed Programs "To achieve a certain degree of confidence that a given program follows its specification, a testing phase must be included in the program development process, and also a complementary debugging phase, to help locating the program's bugs. This paper presents an environment which results of the composition and integration of two basic tools: STEPS (a testing tool) and DDBG (a debugging tool). The two tools are presented individually as stand-alone tools, and we describe how they were combined through the use of another intermediate tool. We claim that the result achieved is a very effective testing and debugging environment."

[LC98b] A Thread-level Distributed Debugger "In order to address the diversity of existing parallel programming models, it is important to provide de-
velopment environments that can be incrementally extended with new services. Concerning the debugging of process-based models, we have previously designed and implemented a basic interface that can be accessed by other tools as well as by debugging modules associated with high-level programming languages.

In this paper we describe our work towards the support of further debugging functionalities for parallel and distributed programs, by describing an interface consisting of basic thread-based debugging commands. We then show how that interface is supported as a new service on top of a distributed monitoring and control software architecture.

[LC98a] The PDBG Process-level Debugger for Parallel and Distributed Programs "In this paper we discuss several issues concerning the design and implementation of a debugger for parallel and distributed applications. This debugger uses a client-server approach to isolate the debugging user-interface from the debugging services, by way of a two-level structured approach: the component-level to observe and act upon individual processes; and the coordination-level to observe the interprocess relations and act upon them.

A formal specification of the expected behavior of a process under debugging is presented, and the debugging functionalities required to support this model are evaluated and its implementation described. Special relevance is given to the tool-interfacing support functionalities of the debugger and how they were implemented."

[CML+98] The DOTPAR Project: Towards a Framework Supporting Domain Oriented Tools for Parallel and Distributed Processing "We discuss the problem of building domain oriented environments by a composition of heterogeneous application components and tools. We describe several individual tools that support such environments, namely a distributed monitoring and control tool (DAMS), a process-based distributed debugger (PDBG) and a heterogeneous interconnection model (PHIS). We discuss our experience with the development of a Problem Oriented Environment in the domain of genetic algorithms, obtained by a composition of heterogeneous tools and application components."

[CLV+98] A Framework to Support Parallel and Distributed Debugging "We discuss debugging prototypes that can easily support new functionalities, depending on the requirements of high-level computational models, and allowing a coherent integration with other tools in a software engineering environment. Concerning the first aspect, we propose a framework that identifies two distinct levels of functionalities that should be supported by a parallel and distributed debugger using: a process and thread-level, and a coordination level concerning sets of processes or threads. An incremental approach is used to effectively develop prototypes that support both functionalities. Concerning the second aspect, we discuss how the interfacing with other tools has influenced
the design of a process-level debugging interface (PDBG) and a distributed monitoring and control layer called (DAMS)."

[CLD98] Tool Integration Issues for Parallel and Distributed Debugging "This paper describes our experience with the design and implementation of a distributed debugger for C/PVM programs within the scope of the SEPP and HP CTI Copernicus projects. These projects aimed at the development of an integrated parallel software engineering environment based on a high-level graphical parallel programming model (GRAPNEL) and a set of associated tools supporting graphical edition, compilation, simulated and real parallel execution, testing, debugging, performance monitoring, mapping, and load balancing. We discuss how the development of the debugging tool was strongly influenced by the requirements posed by other tools in the environment, namely support for high-level graphical debugging of GRAPNEL programs, and support for the integration of static and dynamic analysis tools. We describe the functionalities of the DDBG debugger and its internal architecture, and discuss its integration with two separate tools in the SEPP/HP CTI environment: the GRED graphical editor for GRAPNEL programs, and the STEPS testing tool for C/PVM programs."

[CD98] Monitoring PVM Programs Using the DAMS Approach "Monitoring tools are fundamental components of a development environment as they provide basic support for performance evaluation, debugging, and program visualization. In this paper we describe our experiments with several monitoring tools for the PVM system, namely XPVM, developed at ORNL, Tape/PVM, developed at IMAG Lab., and DAMS, developed at UNL. These tools are compared and their use is described to support instrumentation and monitoring of a high level distributed programming language, PVM-Prolog, a distributed extension to Prolog which provides a full interface to the PVM environment. This language is being used for the implementation of distributed multi-agent applications, and it provides support for heterogeneous multi-lingual programs, built from C and Prolog components which can communicate using PVM primitives."

[CLA98] An Experiment in Tool Integration: the DDBG Parallel and Distributed Debugger "To provide high-level graphical support for PVM (Parallel Virtual Machine) based program development, a complex programming environment (GRADE) is being developed. GRADE currently provides tools to construct, execute, debug, monitor and visualise message-passing parallel programs. It offers high-level graphical programming abstraction mechanisms to construct parallel applications by introducing a new graphical language called GRAPNEL. GRADE also provides the programmer with the same graphical user interface during the program design and debugging stages. A distributed debugging engine (DDBG) assists the user in debugging GRAPNEL programs on distributed memory computer architectures. Tape/PVM and PROVE sup-
port the performance monitoring and visualization of parallel programs developed in the GRADE environment.

[KCD+97] A Graphical Development and Debugging Environment for Parallel Programs "This paper discusses the development of a debugging tool for parallel programs showing how the requirements posed by high-level tools for parallel program development have influenced the design of the debugging system since its early stages of development. We concentrate our attention upon the interfacing of the debugger with other tools of a parallel software engineering environment, namely a graphical programming language and a testing and debugging tool. This is illustrated with the results of our experimentation with the design and implementation of DDBG, a debugger for the PVM environment."

[CDLM98] A Software Architecture for the Integration of Monitoring, Debugging and Profiling Tools for Parallel Program Development "In this paper we present our contribution towards building tools that support the development of parallel and distributed applications composed by heterogeneous components. This research tries to give a positive answer to the needs arising in many application domains where it is necessary to build systems from separately developed components such as visualization tools, interactive control components, virtual reality interfaces, and simulation components. For such environments, software engineering tools can be considered at two distinct levels: the intracomponent level, which corresponds to tools acting upon individual components, and the metacomponent level, that must address the abstractions for the specification of the application configuration and the components interconnection. The distinct aspects of program analysis, including verification, testing and debugging, monitoring and profiling, visualization and user interaction, also appear at the metacomponent level. We describe our approach to build an integrated development environment where the above functionalities can be incrementally included. In this paper we illustrate the current status of our work by describing a prototype environment with a basic monitoring and control layer, and how this was extended to support profiling and debugging services."

[MC97a] Interconnecting Multiple Heterogeneous Parallel Application Components "We present an infrastructure for building parallel applications by interconnecting slightly modified pre existing parallel components. This infrastructure (called PHIS) allows the cooperation of components that run in different parallel machines. In succession, we describe the rationale behind PHIS, the primitives used to interconnect the application components and its internal architecture and we compare PHIS to related systems. Finally, we present an application where PHIS is used to interconnect several distinct components that define a parallel heterogeneous computational steering architecture for genetic algorithm applications."
[MC97b] **Interconnecting Heterogeneous Parallel Applications**  “A distributed heterogeneous information system must include an infrastructure that allows the transfer of information between the different components of an application. In this paper, we present an infrastructure (called PHIS) that supports such kind of facilities when the components of the application are themselves parallel programs.

PHIS makes it possible to develop each component independently from the others and to reuse already existent software, as it supplies a library that supports the interconnection of slightly modified pre-existing parallel components.

The major advantage of PHIS over related systems is the support of communication between applications written using different parallel programming models, e.g. a PVM-based component and a MPI-based component. This contrasts with current message-passing systems (like PVM or MPI-1) where there are almost no facilities for this kind of functionality.

In this paper we discuss the PHIS model, present a simplified example of its use, and give an overview of the implementation. We show how we are using PHIS as an infrastructure to support the development of an interactive steering environment for the parallel execution of genetic algorithms.”

### A.3 Application to Diagnosis

[Sch97] **“Autonomous, Model-based Diagnosis Agents”** This book emerged from a PhD project carried out at University of Hannover, Germany and New University of Lisbon, Portugal. After getting in contact with our Portuguese partners during my master’s thesis, I spent a year in Lisbon and finished my PhD two years later in Hannover. It was a great experience to be involved in several projects carried out at the institutes. In the Prolopp and Padipro projects, the Portuguese side, led by Prof. Pereira, focused on logic programming including issues such as new semantics, distribution, and applications. The German side, led by Prof. Nejdl, has been dedicated to model-based diagnosis and participated in the European project ModelAge to develop a common model of agents. A tedious task. Both institutes cooperated on logic programming for model-based diagnosis over a period of four years.

Based on this background, the objective of this work is the definition and implementation of an architecture for autonomous, model-based diagnosis agents. In this book, we first develop a logic programming approach for model-based diagnosis and introduce strategies to deal with more complex diagnosis problems. Then we embed the diagnosis framework into the agent architecture of vivid agents.

First, we survey extended logic programming and show how this expressive language is used to model diagnosis problems stemming from applications such as digital circuits, traffic control, integrity checking of a chemical database, alarm-correlation in cellular phone networks, diagnosis of an automatic mirror furnace, and diagnosis of communication protocols. To compute diagnoses we review a bottom-up algorithm to remove contradictions from extended logic programs and substantially improve it by top-down evaluation of extended logic.
To deal with complex diagnosis problems we lift the idea of model-based
diagnosis to the meta-level of the diagnostic process and define a strategy
language that allows a declarative description of the diagnostic process. Taking
into account both practical needs and rigorous formal treatment, we define syn-
tax and declarative and operational semantics of the strategy language. With
the concept of deterministic and non-deterministic as well as monotonic and
non-monotonic strategies, we design a strategy knowledge base for circuit diag-
nosis with strategies for structural refinement, choice of models, measurements,
and preferences. We evaluate the knowledge base and the algorithm on a voter
circuit which is part of the benchmark circuits.

Based on the inference engine lined out above we turn to the autonomous
agent's behaviour specification. We present the concept of vivid agents which
comprise a vivid knowledge system and reaction and action rules to specify the
agent's reactive and pro-active behaviour. To realize vivid agents we develop
an architecture for concurrent action and planning. For implementation we use
PVM-Prolog that provides coarse-grain parallelism to spawn agents in a net-
work and fine-grain parallelism to run action and planning component concur-
tently. The interpreter is evaluated in distributed diagnosis where we implement
fault-tolerant diagnosis and diagnosis of a communication protocol. The agent
interpreter satisfies the requirements for a state-of-the-art multi-agent program-
ning language: it supports reactive and pro-active behaviour specification; the
specifications are executable; the language has a formal semantics; the modular
design facilitates plug and play according to the problem domain; the system is
open to heterogeneous agents based on other concepts and languages.

This comprehensive in-depth study of concepts, architecture, and implemen-
tation of autonomous, model-based diagnosis agents will be of great value for
researchers, engineers, and graduate students with a background in artificial in-
telligence. For the practitioners it provides three main contributions: first, many
examples from diverse areas such as alarm correlation in phone networks to in-
consistency checking in databases; second, an architecture to develop agents;
and third, a sophisticated and declarative implementation of the concepts and
architectures introduced. The theorist can benefit from the three contributions
of a novel approach to diagnosis based on logic programming, a newly devised
modal strategy language to model diagnostic processes, and a practical and
formally underpinned concept of agents.

[SW98] "Vivid Agents: Theory, Architecture, and Applications" Vivid
agents are software-controlled systems whose state comprises the mental com-
ponents of knowledge, perceptions, tasks, and intentions, and whose whose be-
haviour is represented by means of action and reaction rules. We present an
execution model for vivid agents which is based on an architecture for concu-
rent action and planning (CAP). We evaluate the concept of vivid agents and
the CAP architecture in distributed diagnosis including fault-tolerant diagnosis

24
and the diagnosis of an unreliable communication protocol.

[SW99] “Distributed Diagnosis by Vivid Agents” Many systems, such as large manufacturing systems, telecommunication networks, or home automation systems, require distributed monitoring and diagnosis. In this article, we introduce a meta-logic interpreter for vivid agents which allows to develop distributed monitoring and diagnosis systems consisting of a variety of scalable knowledge- and perception-based agents. The interpreter is based on PVM-Prolog, an extension of standard Prolog with message passing facilities. We show how to specify and run vivid diagnosis agents carrying out fault-tolerant diagnosis of a computer network.

[FdAMNS97] “Diagnostic Agents for Distributed Systems” In this paper we introduce an agent based framework for the diagnosis of spatially distributed technical systems, based on a suitable distributed diagnosis architecture. We implement the framework using the concepts of vivid agents and extended logic programming. To demonstrate the power of our approach, we solve a diagnosis example from the domain of unreliable datagram protocols.

[SdAMP96] “A Deliberative and Reactive Diagnosis Agent based on Logic Programming” In this article we formally specify and implement a diagnostic agent based on extended logic programming. Motivated by the application of decentralized diagnosis of distributed systems we develop an architecture for such agents that consists of a deliberative layer with a knowledge base, an inference machine and a reactive layer for communication and control. Throughout the layers we employ logic and logic programming to solve these tasks: the knowledge base uses extended logic programming to specify the agent’s behaviour and its knowledge about the system to be diagnosed. The inference machine, which provides algorithms to compute diagnoses, as well as the reactive layer, that realizes a meta interpreter for the agent behaviour, are implemented in PVM-Prolog, with enhances standard Prolog with message passing facilities.

[dAMA95] “Diagnosis of Distributed Systems using Logic Programming” The evolution of logic programming semantics has included the introduction of an explicit form of negation, beside the older implicit (or default) negation typical of logic programming. For the richer language, called extended logic programming, much theoretical work has been done. Mainly resulting from the theoretical work, the language has been shown adequate for a space of knowledge representation and reasoning forms. However, the theoretical work has not been accompanied by the usage of the language for building real-life implementations. In this paper we report on the experience of using extended logic programming to model the diagnosis system. By using extended logic programming, we can rely on a well established language, with a clear declarative semantics, and for which implementations exist. Moreover, some issue of the
diagnosis process are automatically dealt by the contradiction removal methods developed for extended logic programming. Due to the greater expressive power of extended logic programming, the process of implementing diagnosis of distributed systems has been quite simplified. The form of the resulting program is rather simple and has a very clear and declarative reading. Thus, we deem that extended logic programming can be used in practice to solve some “real-life” problems.