

JORNADAS DE SEGUIMIENTO

**PROYECTOS EN TECNOLOGÍAS DE LA  
INFORMACIÓN**

**DESCRIPCIÓN DE RESULTADOS**

**Referencia del proyecto:** TIC99-1151

**Título:** EDIPIA: Entorno de Desarrollo Integrado de Programas  
basado en la Interpretación Abstracta

**Investigador principal:** Francisco Bueno Carrillo

**Dirección de contacto:** Facultad de Informática, UPM,  
Campus de Montegancedo  
28660, Boadilla del Monte, Madrid  
bueno@fi.upm.es

**Datos sobre el grupo investigador:** Grupo CLIP  
Computational Logic, Implementation, and Parallelism  
Computación Lógica, Implementación y Paralelismo

Francisco Bueno Carrillo  
Daniel Cabeza Gras  
Manuel Carro Liñares  
Jesús Correas Fernández  
José Manuel Gómez Pérez  
Pedro López García  
Germán Puebla Sánchez

**¿Se trata de un proyecto coordinado?:** No

# 1 PROJECT OBJECTIVES

The development of correct and efficient software for complex applications is a difficult task. This project aims at an interactive software development environment that allows to find incorrectness errors and inefficiency problems in programs, automatically as much as possible, and during both the initial development phase as well as during the subsequent maintenance of the program.

Errors are detected in the proposed environment from the comparison of specifications, optional and possibly partial, of the program, expressed in a flexible assertion language, with the results of the program execution or an analysis of it. Debugging is thus supported both at run-time and at compile-time, but preference should be given to the latter. The main novelty is in the use of advanced abstract interpretation-based techniques for program analysis, and of safe approximations of the program behaviour, for debugging. This allows to manipulate a much richer set of program properties than those traditionally used, such as type systems. Our properties include not only types but also determinism, non-failure, bounds on the cost of execution and on the size of data, modes, independence of data structures, safety in the data access, etc. The analysis techniques allow to infer such properties, and then the inferred properties can be compared with the specifications. These also allow to document the program in an automated way.

The system should be modular, in the sense of analyzing, compiling, and debugging one module at a time, but allowing to combine the results globally into the final debugged program. This will facilitate the scalability of the method. It is being incorporated into a practical logic-based constraint programming system, Ciao, as part of its program precompiler. Full integration into a single system will allow to combine debugging capabilities and other techniques, such as program optimization, specialization, parallelization, and the like, not necessarily aimed at program debugging.

## 2 LEVEL OF SUCCESS

### 2.1 The source language

We have designed a module system for the language so that it is amenable to modular global analysis, program transformation, and debugging. The main difficulties with existing module systems were in the lack of a clear definition of what the module text is: compile-time expansions, undeclared dynamic predicates, meta-predicates, calls outside the module text, etc., all are characteristics that obscure the process of identifying the code that has to be analyzed. The problems for global analysis were already identified in [1], where they are discussed in detail. The solutions adopted in the Ciao module system [4, 5] include: having the syntax extensions, flags, etc., local to each module; to statically define the module entry points; to use module qualification only for disambiguating predicate names, not for changing naming context; to clearly reference the module text that can be added to a module; to isolate as much as possible the dynamic parts of the code; to view most “builtins” as library module predicates; to clearly distinguish between directives and queries; to declare the exported meta-predicates of a module, etc.

In addition, the module system has made provisions for separate compilation, extensibility in features and in syntax, enhanced error detection, support for meta-programming and higher-order, etc. Of these, the enhanced error detection that a consistently defined module system provides is of direct interest to the project objectives.

From the module system proposed, we have been found that the concept of a module lays the foundation for a very nice approach to agent programming. To include this in the language, we have first provided distributed execution capabilities. We have designed and included in the language low level primitives for concurrency and distribution, and also high level language constructs for this form of programming [8]. We have also evolved the module concept into a model of *active modules* [6], which allows for a very high level, abstract view of distributed execution (and agent programming). We are currently studying the extension of this scheme in two different, but complementary, directions. On one hand, at a higher level, it can be oriented towards component-

based programming [14]. On the other hand, at a finer grain level, we are also considering distributed execution of procedure calls and of objects (and object mobility) [12]. Related to distributed execution, we have developed and implemented a granularity control system [27] based on the same program analysis and transformation scheme of our overall approach. This is a first step towards “resource aware” programming: the code is annotated so that it checks if the distributed execution will pay off w.r.t. the cost of the computation that is going to be distributed. Task granularity control started from [16], but we had problems using the approach originally sketched there of computing complexity functions and performing program transformations at compile-time based on such functions. Taking this approach as a starting point, we have proposed a general granularity control system model [28, 29], able to use information on lower/upper bounds on cost of procedures, and have particularized such model to the case of logic programming and to or/and parallelism [20].

## 2.2 The assertion language

The main aim in our design of an assertion language has been flexibility. Assertions may be used in different contexts and for different purposes: run-time and compile-time checking, providing information to a program optimizer, replacing the oracle in error diagnosis, etc. Many, if not all, of them are related to program debugging. In addition, we also wanted to generate documentation automatically from the program source (in the “literate programming” style suggested by Knuth) based in part on the information present in the assertions. Assertions may express properties which should hold (intended properties) or properties which actually hold (actual properties) for the program. They can express properties of predicates, literals, and of program points. The set of properties of interest themselves can be very wide. Rather than having different assertion languages for each purpose, we have proposed the use of the same assertion language for all of them.

Clearly, the resulting assertion language [31] includes types, but it is more general than type systems in that it allows expressing properties which are often not expressible in customary type systems and which are possibly undecidable. Assertions are optional, not mandatory, and they are used in debugging by each tool as much as it can get benefits from the information conveyed. We argue that our assertion language is a good compromise w.r.t. the clear trade-off between the expressive power of the language of assertions and the difficulty of dealing with it.

The language proposed is parametric w.r.t. the constraint domain and the particular CLP platform being used and thus can be used for any of them. This has made possible to parameterize the debugging tools w.r.t. the source language, since assertions can be used for describing the language builtins for each particular case. The design has also taken into account the possibility of describing conditions for abstractly executing predicates [33], which is useful for program optimization (and, to some extent, for debugging, too). It has also allowed the design and implementation of a program documenter [21, 22] from the assertions.

## 2.3 The analysis component

Based on the abovementioned module system and assertion language, a modular analysis scheme has been designed that allows the communication of predicate properties and module entry points incrementally among the modules of the program being analyzed [34, 2]. The scheme also has provisions for program optimization based on the information gathered. Similar provisions can be made for program debugging, which are currently under investigation.

The analysis component has been extended from the original PLAI analyzer by incorporating to it several new analysis domains (and frameworks). We have focused mainly in type analysis, because of the nice advantages of types for program debugging. The first analyzer based on the regular approximation approach of Gallagher and De Waal has been extended from the predicate level to the level of program points, which greatly improves its benefits for automated debugging (and a limited form of error diagnosis) [25]. Several other alternatives have been studied for improving

the precision of type analyses. The problem here is that the type domain is usually infinite (which is unavoidable if types are to be of any use), and this prevents termination of the analysis unless special provisions are done: this is known as the widening operation. Such operation amounts to a loss of precision. The solution comes, in our opinion, from accurately taking track of the structure of the program, which induces the structure of the types of the program data. We have investigated set-based analysis, which seems promising in this line, and also other types of widening for the more classical analysis frameworks. We have ended up with an alternative technique for set-based type analysis [19], on one hand, and, on the other hand, with a new class of widenings that improve the precision of those previously proposed in the literature [35].

Another (new) analysis that has been designed is safety analysis. The problem to solve here is to identify program fragments that may threaten the safety of a system during execution (by, e.g., accessing protected data). The difficulty for this kind of analysis is in the variable aliasing feature of logic and constraint languages. We have devised an analysis scheme that is based on previously existing variable sharing analyses and is parametric w.r.t. a notion of safety: it identifies variables which can be unsafe by propagating this characteristic via the identified aliasing between variables [3].

We have also developed an analysis of task costs. The property of cost of a computation is useful in performance debugging and also in determining computation overheads for distributed programming (and in task granularity control). We have completed the upper bound cost estimation performed by the CASLOG system [17]. In order to achieve this, we have integrated the system into ours in such a way that the required information on types, modes, and data size metrics is automatically supplied by our analyzers, via the assertion language, thus allowing that the cost analysis be fully automatic. We have also developed an analysis that infers lower bounds on the cost of procedures as functions of input data size. We have implemented this cost analysis and integrated it in our system [24, 23]. The main problem with the inference of lower bounds on the computational cost of logic programs is the possibility of failure. Any attempt to infer lower bounds has to contend with the possibility that a goal may fail during head unification, yielding a trivial lower bound of 0. For this reason, we have had to develop a non-failure analysis able to infer which calls and procedures will not fail, as precisely as possible. We have performed experiments that show that the analysis is accurate (more accurate than the existing ones) and efficient.

The non-failure analysis is based on detecting, for each program predicate, a subset of its clauses which define a *test* on the predicate arguments that *covers* the input type of the calls to the predicate. We have devised correct and complete algorithms for determining precise coverings which are efficient in practice. Several analysis algorithms have been studied for the propagation of the covering information based on fixpoint computations over the program call graph. Covering information can also be used to define and detect determinism of procedure calls. We have adapted the algorithms to obtain an analysis of determinism that we have also incorporated to the system [25].

## 2.4 The debugging component

Assertions should be checked as much as possible at compile-time via static analysis. The inference system should be able to make conservative approximations in the cases in which precise information cannot be inferred (and some assertions may remain unproven). We have devised an assertion checking framework for program debugging along these guidelines [30, 32] that includes both static compile-time checking and dynamic run-time checking. Assertions are compared with analysis results, the assertions that cannot be completely proven nor disproven are simplified (since some of the properties stated could have been proven or disproven, nevertheless) and the program is annotated to check them at run-time. The basic problem here is to handle the properties the assertions talk about: how to define them and how to check them statically and dynamically. Since we aim at an assertion language as generic as possible, and given that our underlying language is a (constraint) logic language, it seems a good compromise to allow to define the properties using the full underlying language. As a result, the properties defined may not be statically decidable.

On the other hand, since they are in fact programs, they can be executed (following the view of logic programs in as “executable specifications”). However, to guarantee that they are dynamically decidable some basic restrictions are imposed on the kind of logic programs that can act as properties for run-time checking. Additionally, the programs defining properties can also be analyzed in order to have a safe approximation of the property: although this cannot lead to proving the property, it can allow to disprove it (by proving that its approximation does not hold).

Given the overall modular approach to debugging, the framework needs be incremental, since modules are handle one at a time, and information is communicated among the processing of each of them. This scheme requires incremental updates of the information for each module whenever the module code, its specification, and/or the information on other modules change. We have started by extending the analysis component to include the possibility of performing incremental analysis [26]. The results have shown that incremental analysis may pay off in terms of efficiency even in the cases where incrementality is not really required. We have also started to study the coupling of incrementality and modularity [34]. There are several possibilities: both the analysis of a module and the modular scheme that merges globally the results for each module can be done incrementally. This gives us a four point space of possible frameworks. We are currently studying the possible advantages and disadvantages of each of them for the purpose of debugging.

An important dimension of interactive debugging is visualization of the program execution. Although it was not included in the original plans of the project, we have also been studying visualization abstractions and algorithms [13, 11], as part of the overall debugging framework. We have defined and implemented several visualization tools for logic and constraint logic program execution [10, 9], which are oriented towards both correctness and efficiency debugging, i.e., to help the programmer in understanding both the program results and its performance.

## **3 RESULTS**

### **3.1 Personnel**

The project has favoured the research activity of at least five members of the group. We have incorporated two new members that are active in the project, both post-graduate students, and several under-graduate students. The project has also resulted in the development of three thesis, two of them already defended [27, 7] and a third one in preparation.

### **3.2 Publications**

We have published 8 conference papers, 3 journal papers, 4 papers in journal special issues, several chapters of a book, and a number of workshop contributions, as well as technical reports, related to the project. A full list of related publications is in the appendix to this document, which serves also as a reference section to the text in this report.

We can also point out two invited talks related to the project: one in the International Conference on Logic Programming, ICLP’99, Las Cruces, New Mexico, (November, 1999), and another one in the International Conference on Computational Logic, CL2000, London (July, 2000), in which the project objectives and advances were presented. Two other invited talks are also scheduled in the near future: one in the 8th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, LPAR’01, in Havana, and another one in the 10th Portuguese Conference on Artificial Intelligence, Porto, both in December, 2001. The organization of two workshops [18, 15] has also helped the widespreading of the project results.

### **3.3 Collaborations**

The work carried out in the scope of the project has given rise to opportunities for collaborations with researchers and research groups at University of Melbourne (Australia), University of Bristol (U.K.), and Universidad Nacional del Comahue (Argentina).

## 4 APPENDIX: Publications (and References)

- [1] F. Bueno, D. Cabeza, M. Hermenegildo, and G. Puebla. Global Analysis of Standard Prolog Programs. In *European Symposium on Programming*, number 1058 in LNCS, pages 108–124, Sweden, April 1996. Springer-Verlag.
- [2] F. Bueno, M. Garcia de la Banda, M. Hermenegildo, K. Marriott, G. Puebla, and P. Stuckey. A model for inter-module analysis and optimizing compilation. In *Tenth International Workshop on Logic-based Program Synthesis and Transformation*, number 2042 in LNCS, pages 86–102. Springer-Verlag, 2001.
- [3] F. Bueno, M. Hermenegildo, G. Puebla, and P. J. Stuckey. Safety for logic programs. Technical Report CLIP1/01.1, Facultad de Informática, UPM, 2001.
- [4] D. Cabeza and M. Hermenegildo. A New Module System for Prolog. In *International Conference on Computational Logic, CL2000*, number 1861 in LNAI, pages 131–148. Springer-Verlag, July 2000.
- [5] D. Cabeza and M. Hermenegildo. The Ciao Modular, Standalone Compiler and Its Generic Program Processing Library. In *Special Issue on Parallelism and Implementation of (C)LP Systems*, volume 30 of *Electronic Notes in Theoretical Computer Science*. Elsevier - North Holland, March 2000.
- [6] D. Cabeza and M. Hermenegildo. Distributed WWW Programming using (Ciao-)Prolog and the PiLLoW Library. *Theory and Practice of Logic Programming*, 1(3):251–282, May 2001.
- [7] M. Carro. *Some Contributions to the Study of Parallelism and Concurrency in Logic Programming*. PhD thesis, Universidad Politécnica de Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, November 2001.
- [8] M. Carro and M. Hermenegildo. Concurrency in Prolog Using Threads and a Shared Database. In *1999 International Conference on Logic Programming*, pages 320–334. MIT Press, Cambridge, MA, U.S.A., November 1999.
- [9] M. Carro and M. Hermenegildo. Tools for Constraint Visualization: The VIFID/TRIFID Tool. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 253–272. Springer-Verlag, September 2000.
- [10] M. Carro and M. Hermenegildo. Tools for Search Tree Visualization: The APT Tool. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 237–252. Springer-Verlag, September 2000.
- [11] M. Carro and M. Hermenegildo. Diseño de visualizaciones para programación lógica con restricciones. (150), March 2001.
- [12] M. Carro and M. Hermenegildo. Remote execution and mobile objects in ciao prolog. Technical Report CLIP3/01.1, Facultad de Informática, UPM, 2001.
- [13] M. Carro and M. Hermenegildo. Visualization designs for constraint logic programming. 2(2), April 2001. Also in UPGRADE. Available through <http://www.svifsi.ch/revue/>.

- [14] J. Correas and F. Bueno. A configuration framework to develop and deploy distributed logic applications. In *ICLP01 Colloquium on Implementation of Constraint and Logic Programming Systems*, Cyprus, November 2001.
- [15] I. de Castro Dutra, V. Santos Costa, G. Gupta, E. Pontelli, M. Carro, and P. Kacsuk (editors). *Parallelism and Implementation Technology for (Constraint) Logic Programming*, volume 30 of *Electronic Notes in Theoretical Computer Science*. Elsevier - North Holland, Elsevier Science, P.O. Box 211, 1000 AE Amsterdam, The Netherlands, March 2000.
- [16] S.K. Debray, N.-W. Lin, and M. Hermenegildo. Task Granularity Analysis in Logic Programs. In *Proc. of the 1990 ACM Conf. on Programming Language Design and Implementation*, pages 174–188. ACM Press, June 1990.
- [17] S.K. Debray and N.W. Lin. Cost analysis of logic programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, November 1993.
- [18] M. Ducassé, A. Kusalik, and G. Puebla, editors. *Logic Programming Environments*, volume 30 of *Electronic Notes in Theoretical Computer Science*, Elsevier Science, P.O. Box 211, 1000 AE Amsterdam, The Netherlands, March 2000. Elsevier - North Holland.
- [19] J. Gallagher and G. Puebla. Abstract Interpretation over Non-Deterministic Finite Tree Automata for Set-Based Analysis of Logic Programs. In *Fourth International Symposium on Practical Aspects of Declarative Languages*, LNCS. Springer-Verlag, January 2002. Accepted for publication.
- [20] G. Gupta, E. Pontelli, K. Ali, M. Carlsson, and M. Hermenegildo. Parallel Execution of Prolog Programs: a Survey. *ACM Transactions on Programming Languages and Systems*, 23(4):1–131, July 2001.
- [21] M. Hermenegildo. A Documentation Generator for (C)LP Systems. In *International Conference on Computational Logic, CL2000*, number 1861 in LNAI, pages 1345–1361. Springer-Verlag, July 2000.
- [22] M. Hermenegildo. A System for Automatically Generating Documentation for (C)LP Programs. In *Special Issue on Parallelism and Implementation of (C)LP Systems*, volume 30 of *Electronic Notes in Theoretical Computer Science*, March 2000.
- [23] M. Hermenegildo, F. Bueno, D. Cabeza, M. Carro, M. García de la Banda, P. López-García, and G. Puebla. The CIAO Multi-Dialect Compiler and System: An Experimentation Workbench for Future (C)LP Systems. In *Parallelism and Implementation of Logic and Constraint Logic Programming*, pages 65–85. Nova Science, Commack, NY, USA, April 1999.
- [24] M. Hermenegildo, F. Bueno, D. Cabeza, M. Carro, M. García de la Banda, P. López-García, and G. Puebla. The Ciao Logic Programming Environment. In *International Conference on Computational Logic, CL2000*, July 2000.
- [25] M. Hermenegildo, F. Bueno, G. Puebla, and P. López-García. Program Analysis, Debugging and Optimization Using the Ciao System Preprocessor. In *1999 International Conference on Logic Programming*, pages 52–66, Cambridge, MA, November 1999. MIT Press.
- [26] M. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems*, 22(2):187–223, March 2000.
- [27] P. López-García. *Non-failure Analysis and Granularity Control in Parallel Execution of Logic Programs*. PhD thesis, Universidad Politécnica de Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, June 2000.

- [28] P. López-García, M. Hermenegildo, and S.K. Debray. Towards Granularity Based Control of Parallelism in Logic Programs. In Hoon Hong, editor, *Proc. of First International Symposium on Parallel Symbolic Computation, PASC0'94*, pages 133–144. World Scientific, September 1994.
- [29] P. López-García, M. Hermenegildo, and S.K. Debray. A Methodology for Granularity Based Control of Parallelism in Logic Programs. *Journal of Symbolic Computation, Special Issue on Parallel Symbolic Computation*, 22:715–734, 1996.
- [30] G. Puebla, F. Bueno, and M. Hermenegildo. A Generic Preprocessor for Program Validation and Debugging. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 63–107. Springer-Verlag, September 2000.
- [31] G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Constraint Logic Programs. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 23–61. Springer-Verlag, September 2000.
- [32] G. Puebla, F. Bueno, and M. Hermenegildo. Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs. In *Logic-based Program Synthesis and Transformation (LOPSTR'99)*, number 1817 in LNCS, pages 273–292. Springer-Verlag, 2000.
- [33] G. Puebla and M. Hermenegildo. Abstract Multiple Specialization and its Application to Program Parallelization. *J. of Logic Programming. Special Issue on Synthesis, Transformation and Analysis of Logic Programs*, 41(2&3):279–316, November 1999.
- [34] G. Puebla and M. Hermenegildo. Some Issues in Analysis and Specialization of Modular Ciao-Prolog Programs. In *Special Issue on Optimization and Implementation of Declarative Programming Languages*, volume 30 of *Electronic Notes in Theoretical Computer Science*. Elsevier - North Holland, March 2000.
- [35] C. Vaucheret and F. Bueno. Structural type widening. Technical Report CLIP2/01.1, Facultad de Informática, UPM, 2001.