Why Ciao?
An Overview of the Ciao System's Design Philosophy

Manuel Hermenegildo and
The Ciao Development Team

Our intention in this note is not to provide a listing of the many features of the Ciao system [13,12,2]: this can be found in part for example in the brochures announcing upcoming versions, in the Ciao website, or in more feature-oriented descriptions such as [10]). Instead in this document we would like to describe the objectives and reasoning followed in our design as well as the fundamental characteristics that in our opinion make Ciao quite unique and hopefully really useful to you as a Ciao user.

Prolog? Yes! –"more and less." When you start the Ciao top level and load a standard Prolog program you experience Ciao as a modern Prolog system. And Ciao is certainly a high-quality, high-performance, fully featured, public domain (and widely used!) implementation of Prolog. So, if you were looking for a truly great Prolog you came to the right place!

But Ciao is also much, much more than a Prolog system. It essentially represents our best effort at designing and implementing what we believe a truly "next-generation," multi-paradigm *programming language* and *program development environment* should be.

You may ask then, "If that is your intention, why even bother to support Prolog? All other "next-generation" logic programming systems decided to depart from Prolog and give up on running Prolog programs!" The answer is because, first, as Ciao shows, Prolog can be supported without giving up on having also a truly next-generation system (we explain how below). And second, because supporting Prolog is clearly of practical interest: it is an industry standard and it is the principal logic language taught to students, so there is considerable expertise available in its builtins and intricacies. And Prolog also implements a quite interesting set of design choices, worth having available.

The key issue here is that Ciao supports Prolog *through libraries*. This means that the Prolog features (such as all the builtins or even the depth-first search) are not bolted into the language but rather are *options* that can be loaded or not into a given module at will. The basic substrate of Ciao is in fact a simple, horn clause-based pure kernel language. But this substrate (and the whole system) is designed to be easily and highly extensible via a mechanism called "packages" [5]. This mechanism allows adding new syntax to the language and giving new semantics to this syntax. Most importantly, different extensions can be made active in different modules. Using this mechanism, a set of Ciao packages effectively implements all the functionality of the highest-quality, modern Prolog systems. This set of packages is loaded by default when a Prolog module is read in. Also, the top level starts in this mode. So, yes, you get a truly great Prolog system, and this is quite sufficient for many of Ciao's users.

Logic programming beyond Prolog... But the "building block" nature of the Ciao system allows us to provide a logic programming system that goes well beyond Prolog. In particular one

can easily set up for example a given module (or all modules) to not load any of Prolog's builtins, and thus have a *pure logic language*. One can then load for example a declarative I/O library instead of the Prolog I/O. Furthermore, it is not even necessary to stay within the depth-first, left-to-right execution regime of Prolog: by loading different packages several computation rules (breadth-first, iterative deepening, Andorra model, etc.) are supported. These have shown useful in applications and also very specially when teaching logic programming. In our experience it is cumbersome to make the first introductory lectures to logic programming using Prolog since the particular (albeit often practically useful) quirks and the subsequent non-termination of Prolog get in the way of teaching the fundamental concepts of logic programming. We have found that it makes perfect sense to start with a purer logic language, with better termination and fairness characteristics, and the Ciao breadth-first modes have proved quite useful for this (see `http://www.cliplab.org/logalg` for the slides of our course based on this approach). Once the beauty of pure logic programming is experienced the student is then introduced to the practical and powerful choices made in the design of Prolog, as later to topics and functionality beyond Prolog, such as those outlined below (all within Ciao!).

Packages and more packages... Indeed, a large number of additional packages provide language extensions for several types of constraint programming (clpq, clpr, finite domains, ...), higher-order logic programming (with predicate abstractions), feature terms (records), persistence, answer set programming, etc. Other libraries also support WWW programming, sockets, many external interfaces, etc. The open design of the language facilitates the development of extensions and in fact many of these packages have been contributed by users. Thus, Ciao is not only a language and programming environment but also a language building toolkit.

No point in getting locked in a single programming paradigm... In fact, the extensibility of the kernel language also means that Ciao is a truly *multiparadigm programming system* incorporating many of the best features of other programming paradigms outside logic and constraint programming. In particular, the system allows defining functions, including higher-order (function abstractions) and (optionally) lazy evaluation, as well as using functional syntax for both functions and predicates. It also supports object-oriented programming, concurrency (with concurrent built-in database), distributed execution (agents), and parallel execution. Again, all of these are provided *as libraries* and can be activated or deactivated on a *per-module* basis. As an example, this allows us to teach everything from pure LP, through Prolog, constraints, functional programming, higher order, concurrency, and even some aspects of object-oriented programming using a single system.

Assertions (types, modes, cost, ...): yes, but optional! Many modern languages (such as say, Mercury or Haskell) *require* users to provide type and/or mode declarations for procedures or impose other related requirements such as having to define explicitly all types used or all procedures having to be "well-typed." One argument in favor of such declarations and restrictions is that they can be useful to clarify interfaces and meanings, and in general to make large programs more maintainable and well documented, facilitating "programming in the large".

We certainly agree with this! But at the same time we also wanted Ciao to be useful (as, say, Prolog or Scheme) for "programming in the small," i.e., for prototyping, developing simple scripts, teaching, or simply experimenting while trying to find a solution to a problem, ... and for this we feel type and mode declarations and other related restrictions simply get in the way.

Fortunately, we came up with a very good solution to this apparent conundrum: Ciao does include a very rich assertion language which allows expressing a wide variety of properties (including types, modes, determinacy, non-failure, cost, ...), but in Ciao *these assertions are optional*. This solution has allowed us to design Ciao to be very useful both for programming in the small and in the large. We believe that Ciao's unique solution to the issue of assertions combines effectively the advantages of the strongly typed and untyped language approaches, bringing the best of both worlds to the programmer.

Program Documentation, Static Debugging, and Verification: One of the most useful features of the assertions used in Ciao is that they are designed to serve many purposes. First, any assertions present in programs can be processed by an autodocumenter (`lpdoc` [11]) in order to generate useful documentation. Also, such assertions are analyzed interactively during program development by a preprocessor (`ciaopp` [3,14,15]) which can *find sophisticated bugs* statically, *verify* that the program complies with the assertions, or even generate automatically proofs of correctness that can be shipped with programs and checked easily at the receiving end (using the *proof/abstraction carrying code* approach [1]). Even if a program contains no assertions, Ciao checks the uses that programs make of libraries (which do contain assertions) thus catching additional bugs at compile time. If the system cannot prove nor disprove some property at compile time, it can (optionally again) introduce a run-time check for it in the executable.

Interestingly, the technology that allows the system to cope with assertions not being present for all predicates also allows dealing with much more complex properties (beyond for example simple types and modes) in a safe way. As a result, for example, the programmer has the possibility of stating assertions about the efficiency of the program (lower and/or upper bounds on the computational cost of procedures) which the system will try to verify or falsify, thus performing automatic debugging and validation of the *performance* of programs! And many other interesting properties of the predicates and literals of the program can be handled such as data structure shape (including pointer sharing), bounds on data structure sizes, and other operational variable instantiation properties, as well as procedure-level properties such as determinacy, termination, non-failure, and bounds on the consumption of user-defined resources. Assertions also allow programmers to describe modules that are written in other languages (or are not yet written). Finally. the compiler reports the results of analysis also using the assertion language.

Very high performance of course, but with less effort: The second potential benefit of strongly typed languages is performance: the compiler can generate more efficient code with the additional type and mode information that the user provides. Performance is a good thing, of course, but we do not find it good high-level language design to put the burden of efficient compilation on the user by requiring the presence of many program declarations: the compiler should instead do the work of inferring such program properties. This is again the approach taken in Ciao: when assertions are not present in the program, Ciao's analyzers *infer* them. The objective is again to achieve the best of both worlds: with no assertions or analysis the Ciao compiler (`ciaoc` [6]) generates code which is highly competitive in speed and size with the best untyped systems (e.g., with the best commercial or academic Prolog systems). And then, when useful information is present (either coming from the user or *inferred by the system analyzers*) the (experimental) compiler produces code that is very competitive with that coming from strongly-typed systems.

The information inferred by the global analyzers can be used to perform source-level code optimizations, including multiple abstract specialization, partial evaluation, dead code removal, goal

reordering, parallelization with granularity control, reduction of concurrency / dynamic scheduling, low-level optimization, etc. [15].

Versatile, Incremental Compiler and Abstract Machine: Several types of executables can be built easily. In addition to the traditional Prolog top-level, the system offers support for the use of Ciao as a scripting language, for compilation to multiarchitecture *bytecode* executables, and for compilation to single-architecture, standalone executables. Multiple platforms are supported, including Windows, Linux, Mac Os X, and many other Un*x-based OSs. Optimizing compilation to native code (via C) is in a mature state [9] and will be part of the standard distribution in the near future.

A comparatively simple, but optimized abstract machine supports the kernel language. It includes native support for *attributed variables* and for threading primitives, and a synchronization-enabled shared database [8].

Other support for programming in the small and in the large: In addition to the the handling of assertions programming in the large is further supported by a robust module/class system. This novel design [5] enables modular program development, effective global program analysis, modular static debugging, and module-based automatic incremental compilation and optimization. The compiler performs automatically such incremental modular compilation (i.e., incrementality is achieved without any need to define "makefiles").

Programming in the small is additionally supported by having significantly reduced size executables, which include only those builtins and libraries used by the program, a fully interactive environment with seamlessly integrated interpreter, compiler, and source debugger, and by other features such as supporting Prolog *scripts*.

An Advanced Program Development Environment: another design objective of Ciao has been to provide a truly comfortable program development environment that allows developing correct and efficient programs in as little time and with as little effort as possible. This includes a *rich graphical development interface*, based on the latest, graphical versions of Emacs (offering menu and widget-based interfaces with direct access to the top-level/debugger, preprocessor, and autodocumenter) as well as an embeddable source-level debugger with breakpoints, and several execution visualization tools. The environment provides also automated access to the documentation, extensive syntax highlighting, auto-completion, or auto-location of errors in the source, and is highly customizable. *Automatic Documentation Generation:* the assertions and directives present in the program and libraries, as well as all other program information available to the compiler, are used to generate automatically program documentation (including types, modes, machine-readable comments, etc.) by means of the Ciao autodocumenter [11]. A plugin with very similar functionality is also available for the Eclipse environment.

Rich interfaces to other systems and languages: programs often need to interact with other components. Ciao offers multiple bidirectional foreign interfaces to C (with automatic generation of glue code), Java, TclTk, SQL databases (with a notion of predicate persistence), etc.

Support for Concurrency, Parallelism, and Distributed Execution: Ciao includes concurrency, parallelism, and distributed execution capabilities [4]. The notion of "active module" (or active object) allows compiling modules in such a way that they are ultimately mapped to a standalone process, which is transparently accessed by the rest of the application. In addition, the system also offers a full-fledged library for developing WWW-based applications [7].

Free Availability: Ciao is free software protected to remain so by the GNU LGPL license. It can be used freely to develop both free and commercial applications.

Finally, the system includes a **large set of libraries**.

But why is it called Ciao? After reading the previous paragraphs the sharp reader may have already seen the logic behind the "Ciao Prolog" phrase. Ciao is an interesting word which is used both to say *hello* and *goodbye*. Ciao intends to be a truly excellent, high-performance, and freely available ISO-Prolog system which can be used as a classical Prolog, in both academic and industrial environments (and, in particular, to introduce users to Prolog and to constraint and logic programming –the *hello* Prolog part). But Ciao is also a new-generation, multiparadigm programming language and sophisticated program development environment which goes well beyond Prolog and other classical logic programming languages –the *goodbye* Prolog part. And it has the advantage (when compared to other modern systems) that it does so while keeping full Prolog compatibility when desired.

Probing Further

The reader is encouraged to explore the system, its documentation, and the tutorial papers that have been published on it. We are currently working on the new 1.14 system version which includes significant enhancements with respect to the previous version (1.10), including integration of the preprocessor and autodocumenter into the Ciao development tree as a single package (previously they had to be downloaded and installed separately). This version is available already on demand from the Ciao subversion repository.

Contact / download info:

`http://www.ciaohome.org`
`http://www.cliplab.org`
`ciao@clip.dia.fi.upm.es`

The Ciao Development Team
Technical U. of Madrid, Spain
U. of New Mexico, USA
U. Complutense de Madrid, Spain

Acknowledgments: The Ciao system is in continuous and very active development through the collaborative effort of numerous members of several institutions, including UPM, UNM, UCM, Roskilde U., U. Melbourne, Monash U., U. Arizona, Linköping U., NMSU, K. U. Leuven, Bristol U., Ben-Gurion U, INRIA, as well as many others. The development of the Ciao system has been supported by a number of European, Spanish, and other international projects (currently by EU IST-15905 *MOBIUS* project, the Spanish TIN-2005-09207 *MERIT* project, and the CAM *PROMESAS* program. Manuel Hermenegildo is also supported by the IST Prince of Asturias Chair at the University of New Mexico. The system documentation and related publications contain more specific credits for the many contributors to the system.

References

1. E. Albert, G. Puebla, and M. Hermenegildo. Abstraction-Carrying Code. In *Proc. of LPAR'04*, number 3452 in LNAI, pages 380–397. Springer-Verlag, 2005.
2. F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla (Eds.). The Ciao System. Reference Manual (v1.13). Technical report, School of Computer Science (UPM), 2006. Available at `http://www.ciaohome.org`.
3. F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Maluszynski, and G. Puebla. On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*, pages 155–170, Linköping, Sweden, May 1997. U. of Linköping Press.
4. D. Cabeza and M. Hermenegildo. Distributed Concurrent Constraint Execution in the CIAO System. In *Proc. of the 1995 COMPULOG-NET Workshop on Parallelism and Implementation Technologies*, Utrecht, NL, September 1995. U. Utrecht / T.U. Madrid. Available from `http://www.cliplab.org/`.
5. 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.
6. 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(3) of *Electronic Notes in Theoretical Computer Science*. Elsevier - North Holland, March 2000.
7. 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.
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, USA, November 1999.
9. M. Carro, J. Morales, H.L. Muller, G. Puebla, and M. Hermenegildo. High-Level Languages for Small Devices: A Case Study. In Krisztian Flautner and Taewhan Kim, editors, *Compilers, Architecture, and Synthesis for Embedded Systems*, pages 271–281. ACM Press / Sheridan, October 2006.
10. The Ciao Development Team. The Ciao Multiparadigm Language and Program Development Environment, November 2006. The ALP Newsletter 19(3). The Association for Logic Programming. Available from `http://www.logicprogramming.org/newsletter/nov06/index.html`.
11. 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.
12. 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.
13. M. Hermenegildo and The CLIP Group. Some Methodological Issues in the Design of CIAO - A Generic, Parallel, Concurrent Constraint System. In *Principles and Practice of Constraint Programming*, number 874 in LNCS, pages 123–133. Springer-Verlag, May 1994.
14. M. Hermenegildo, G. Puebla, and F. Bueno. Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In K. R. Apt, V. Marek, M. Truszczynski, and D. S. Warren, editors, *The Logic Programming Paradigm: a 25–Year Perspective*, pages 161–192. Springer-Verlag, July 1999.
15. M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, October 2005.

Most of these and other papers and technical reports related to Ciao can be obtained from the Clip lab main WWW server, `http://www.cliplab.org/`.