Organizational Information Systems Design and Implementation with Contextual Constraint Logic Programming

Salvador Abreu* **and Daniel Diaz**† **and Vitor Nogueira***

* Universidade de Évora and CENTRIA, Portugal
{spa,vbn}@di.uevora.pt

† Université de Paris I and INRIA, France
Daniel.Diaz@univ-paris1.fr

Abstract

In this article we claim that Contextual Constraint Logic Programming (CCxLP for short) is a powerful paradigm in which to design and implement Organizational Information Systems, particularly when integrated with the ISCO/ISTO mediator framework. We briefly introduce the language and its underlying paradigm, assessing it from the angle of its antecedents: Logic and Object-Oriented Programming. To further the point, we focus on the use of implicit temporal information in ISTO and show how this formalism can be used to enhance expressiveness. An implementation has been developed which is being actively used in a real-world setting – Universidade de Évora's second generation web-based Academic Information System, which we briefly report on. We conclude that the risks taken in adopting a developing technology such as this for a mission-critical system has paid off, in terms of both development ease and flexibility as well as in maintenance requirements.

1 Introduction

In the process of devising a strategy for the gradual design and deployment of SIIUE, an organizational information system for Universidade de Évora, we were faced with hard choices: as there are really no available ready-to-use solutions, some measure of in-house development was inevitable: Academic organizations have several specificities which are not well catered for by existing ERP products. The adoption of an existing methodology based on existing (commercial) tools was considered, but deemed to be rather inflexible and ultimately expensive in the long run, moreover, it would hardly build on the known-how acquired through the developments which had been locally initiated over the preceding years.

Our explicitly assumed option was to rely on open-source software as much as possible, in order to avoid vendor lock-in and to ensure that future developments could always be effected using in-house competence. In doing so, we had to be very careful in order to pick the most appropriate tool for each particular aspect of the project, always making sure that everything should fit harmoniously.

The overall coordination of the various software components of such a mixed system was critical to the success of the project: this is where Logic Programming appears to be a very interesting and promising choice. Early work on the design and development of SIIUE [1] already indicated that it would benefit from Logic Programming tools playing an increasingly central part in the system. This observation, later confirmed by further developments [2], led to incrementing the investment in the technology which was at the base of the entire project: Prolog as the foundation of an Organizational Information System specification *and implementation* language.

The benefits of Logic Programming are well known: the rapid prototyping ability and the relative simplicity of program development and maintenance, the declarative reading which facilitates both development and the understanding of existing code, the built-in solution-space search mechanism, the close semantic link with relational databases, just to name a few. The realization that Logic Programming is a promising tool with which to address this type of problem is not exclusive to Universidade de Évora's project, as witness for instance the work described in [14].

Our choice of GNU Prolog as the basic tool with which to develop our system was due to several factors, not the least of which is its inclusion of a complementary problem-solving paradigm: *constraint programming.* Constraints strengthen the declarative programming facet of Prolog, providing an *a-priori* search-space pruning model which complements the *a-posteriori* depth-first backtracking search of classical Prolog implementations of Logic Programming.

Nevertheless, the Prolog language suffers from a serious scalability issue when addressing actual applications. There have been several efforts over the years to overcome this limitation: one which took many years to shape is the ISO standard formulation of *modules* for Prolog. This standard can hardly be considered satisfactory, being heavily influenced by previously existing and conflicting implementations: it can be argued that it essentially introduces the concept of "separate predicate spaces," which are reminiscent of ADA modules. Moreover, the standard is syntactically very verbose, which in itself is a very questionable departure from what we perceive to be one of Prolog's strengths: its syntactic simplicity. Moreover, it is our opinion that the standard completely misses the opportunity it had of assimilating closely related yet well established and vastly more powerful concepts, such as the notions of Object and Inheritance.

An interesting alternative solution to the same problem is that of *Contextual Logic Programming* (CxLP), a model introduced in the late 1980's. Informally, the main point of CxLP is that programs are structured as sets of predicates (*units*) which can be dynamically combined in an execution attribute called a *context*. Goals are seen just as in regular Prolog, except for the fact that the matching predicates are to be located in all the units which make up the current context.

We extended CxLP to attach *arguments* to units: these serve the dual purpose of acting as "unit-global" variables and as state placeholders in actual contexts.

We used GNU Prolog/CX to implement the newer components of SIIUE: the redesigned Academic Services subsystem. This article reports on the outcome of this initiative.

The remainder of the article is structured as follows: in section 2 we describe how contexts constitute a useful extension to Logic Programming, drawing it close to OOP. Section 3 introduces the ISTO language. In section 4 an application – a university's Academic Services management system – is sketched. Section 5 presents a very brief overview of the underlying technology which was developed to build the system on. Section 6 draws some concluding remarks and points at directions for future work.

2 Contexts as Objects with State

The integration of the Object-Oriented and Logic Programming paradigms has long been an active research area since the late 1980's; take for example McCabe's work [8]. The similarities between Contextual Logic Programming and Object-Oriented Programming have been focused several times in the literature; see for instance the work by Monteiro and Porto [10] or Bugliesi [4].

Other than the implementation-centered reports, previous work on Contextual Logic Programming focuses largely on issues such as the policy for context traversal, what the context becomes once a unit satisfying the calling goal is found, what to do when multiple units provide clauses for the same predicate, how to automatically tie several units together or how to provide encapsulation and concealment mechanisms.

To the best of our knowledge, no published work earlier than [3] builds on the notion of context arguments and their widespread use, even though Miller's initial work [9] already mentions the possibility of using module variables. This feature was present as a "hack" in the first C-Prolog based implementation of Contextual Logic Programming but was a little let down, possibly for lack of an adequate formalization and the nonexistence of convincing examples.

Instead of viewing a context as an opaque execution attribute, as happens in CSM [11] for instance, we choose to regard it as a first-class entity, i.e. as a Prolog term. Not only is the context accessible from the program, but *it is intended* that it be explicitly manipulated in the course of a program's regular computation. The performance impact of this option will be succinctly analyzed in section 5: at this point we shall concentrate on the possibilities it allows from an expressiveness point of view, relating Contextual Logic Programming examples to other paradigms whenever appropriate.

2.1 Contexts and Object-Oriented Languages

Table 1 establishes some parallels between Contextual Logic Programming (CxLP) and Object-Oriented Programming (OOP) terminology, pointing out how units, contexts and context arguments can relate to OOP concepts. The most notable difference between the CxLP and the OOP paradigms has to do with the concept of *inheritance*: instead of being statically defined as in the Class-based Object-Oriented languages, it is completely dynamic for each context (i.e. "object"), as it defines its own structure and, implicitly, its behaviour wrt. messages.

OOP (Class)	**OOP (Prototype)**	**CxLP**
Object	Object	Context
Message	Message	Goal
Instance variable	Named slot	Unit argument
Method	Method	Predicate
Class	-	Context skeleton
Class member	-	Unit
Inheritance	Object inheritance	Context traversal
Class instantiation	Object cloning	Context term manipulation

Table 1: CxLP vs. OO paradigms (Class- and Prototype-based)

CxLP enables design approaches stemming from class-based but mostly prototype-based or object-centered languages (examples of which include Self [15] and JavaScript) in that a *unit* can be seen as akin to a class as it defines partial state and behaviour while a *context*, as a self-sufficient object, can serve as the basis for the creation of further contexts, either via the extension mechanism or by explicit manipulation of the context term (e.g. a copy).

2.2 Encapsulation and Concealment

These issues are central in Object-Oriented Programming and critical from the Software Engineering point of view. Earlier approaches in Contextual Logic Programming languages proposed several distinct mechanisms, along the lines of having an annotation of some sort to indicate that a given predicate was to be considered "visible" or "hidden", in the sense that a context traversal would see it or not.

Our approach of relying on deep contexts and unit arguments, made possible by the relative efficiency of the prototype implementation, as described in [3], allows us to shun the introduction of yet another set of predicate annotations, because simpler constructions are effectively available, through the use of unit arguments and the context switch operation: all that is necessary is that the context arguments supply sufficient information for a *new* context to be built, in order to implement the requested method without disclosing the details to the invoking context.

2.3 Contexts as Implicit Computations

ISTO relies on GNU Prolog/CX as its compiler's target language and is further described by means of an application in section 4.

Consider a unit `person(ID, NAME, BIRTH_DATE)` which defines the following predicates:

- `item/0` which returns, through backtracking, all instances of the `person/3` database relation by instantiating unit arguments,
- `delete/0` which nondeterministically removes instances of the `person/3` database relation, as restricted by the unit arguments,
- `insert/0` which inserts new instances into the `person/3` database relation, taking the values from the unit argument.

Accessing an "object" specified by a context is always done via one of these predicates, which are to be evaluated in a context which specifies the relation (in this case `person/3`). Assume that there are also predicates with the same name and one argument, which represents the relevant unit with bound arguments, i.e. `item/1`, `delete/1` and `insert/1`. An implementation of these predicates can rely on the standard Prolog internal database manipulation built-ins or access an external database, as is done in the ISCO compiler [2].

3 A Case for the ISTO Programming Language

ISCO [2] is a Logic Programming language geared towards the development and maintenance of organizational information systems. ISCO is an evolution of the previous language DL [1] and is based on a Constraint Logic Programming framework to define the schema, represent data, access heterogeneous data sources and perform arbitrary computations. In ISCO, processes and data are structured as *classes* which are represented as typed[1] Prolog predicates. An ISCO class may map to an external data source or sink, such as a table or view in a relational database, or be entirely implemented as a regular Prolog predicate. Operations pertaining to ISCO classes include a *query* which is similar to a Prolog call as well as three forms of *update*. These operations are syntactically specified as Prolog goals related to the predicate which stands for a particular class.

Most (all?) life-size applications have time-related aspects: be it historical records or information which is inherently temporal in nature. Traditionally these issues are dealt with in an ad-hoc fashion, by "custom programming" solutions in the implementation language, which have to rely on sometimes unnatural modifications to the application schema.

[1]The type system applies to class members, which are viewed as Prolog predicate arguments.

ISTO is a language which:

- Builds on the potential provided by combining contexts and the ISCO class declarations and
- Effectively supports temporal aspects without burdening the application schema and code.

3.1 Classes in ISTO

ISTO is very similar to ISCO, in that relation declarations are used to produce access predicates, to be used as regular Prolog goals. However, there is a twist, resulting from the inclusion of the Contextual Logic Programming constructs: each class declaration results in the transparent definition of an associated unit, which allows for uniform access, via unique conventioned predicate names associated with the admissible operations.

Contexts are very powerful constructs in that they provide a very natural representation for parametric stored queries, see for instance section 2.3 for a suggestion of what this means.

3.2 Time in ISTO

We are presently evolving ISCO to endow it with an expressive means of representing and implicitly using temporal information [12, 13], the resulting language is called ISTO.

ISTO uses CLP to express the constraints imposed by the different temporal systems (for example: the Gregorian calendar or 24–Hour timekeeping), for the evaluation of temporal expressions or the conditional admissibility of particular tuples as solutions to goals. For instance, to represent the 24–Hour timekeeping system we use the triple $(HOUR, MINUTE, SECOND)$ of finite domain variables and the constraint $0 \leq HOUR \leq 23 \wedge 0 \leq MINUTE \leq 59 \wedge 0 \leq SECOND \leq 59$.

CxLP allows us to encapsulate the concepts stated above, i.e. we have one unit for each temporal system, another for the common temporal expressions, ... For instance, to represent the weekends of January and February, 2004, we write:

```
date(D) :> ( year(2004), month(M), M<3,
             weekday(WD), member(WD, [sat, sun]) ).
```

In this example, we extend the initially empty context with the unit `date`, and argument variable `D` that represents a tuple `(YEAR, MONTH, DAY)`. We subsequently constrain `YEAR` to be equal to 2004 and `MONTH` to be one of the set $[1,2]$, i.e., less than 3. Finally we restrict `DAY` to be a Saturday or a Sunday, i.e. belong to the weekend.

CxLP is also used to associate time with facts. As an example, consider that we want to represent that a given lecturer (John) teaches Logic Programming every Monday from 9 to 11, during the odd semester of 2003/2004. Assuming that there is a unit called `semester` that stores, for each academic year, the start and end of each semester, the intended fact can expressed by:

```
1 semester(odd, 2003) :> valid_time(I),
2 date_time(D) :> ( weekday(mon),
                    hour(H), H >= 9, H <= 11),
3 temporal_expression(member(D, I)) :>  true,
4 class_schedule(john, 'Logic Programming') :> (
     valid_time(D), insert).
```

We start by defining variable `I` as the temporal interval corresponding to the odd semester of 2003/2004. We then specify variable `D` to be the recurring interval between 9AM and 11AM, of every Monday in the calendar. We then restrict `D` to be within the interval `I`. Finally, we insert the intended fact in the `classe_schedule` database.

Now if we ask for all the dates when John teaches

`"?- classes_schedule('John', _) :> (valid_time(D))."`

then variable `D` should contain (at least) all the the Mondays, of the Odd semester of 2003/2004, between 9AM and 11AM.

4 Universidade de Évora's Academic Information System

GNU Prolog/CX has already seen actual use in a real-world application: Universidade de Évora's second generation Academic Information System, which is a project that got under way in March 2003 and, at the time of this writing (October 2003), is already in production. This initiative was spurred by the University's decision to simultaneously reorganize all of its undergraduate offerings, to comply with the "Bologna principles," a goal which could not be met by the existing system without very significant and resource-consuming overhauls.

The Academic Information System (SIIUE.sac) is part of Universidade de Évora's Integrated Information System [1] (SIIUE), being its latest component and a useful and diverse testbed for the ISCO and ISTO languages.

4.1 The Academic Information System; SIIUE.sac

The architecture for the Academic Information System can be summarized by figure 1: the different layers correspond to actual physically different networks, interfacing each pair of layers which have contact.

The physical separation is provided to ensure that access to higher-numbered layers is exclusively performed by hosts on the layer immediately below.

There is the requirement that, since all validation and authorization is performed by the ISTO layer, the layers above only access any application data via ISTO, hence the application must have three layers, as seen in figure 2.

Layers 1 and 2 operate on multiple processors for better

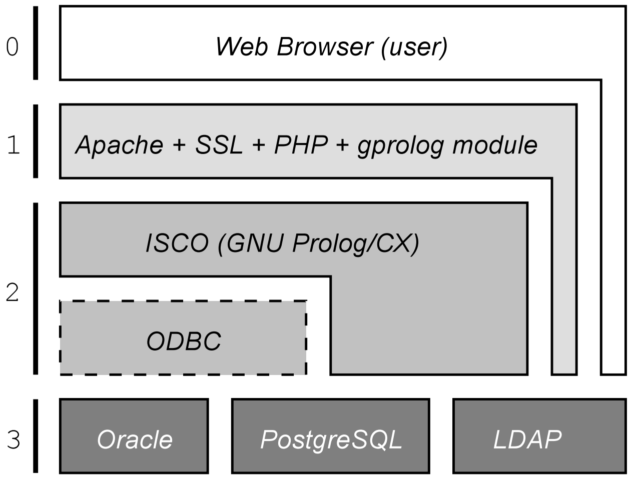

Figure 1: Layers in ISTO

throughput. The ISTO (GNU Prolog/CX executable) processes come from a pool where they perform initialization tasks before becoming available as query processors, thereby bypassing the overhead of some initialization chores, such as connecting to database servers. It should be noted here that GNU Prolog's architecture is very favourable to its usage as a Prolog implementation for this type of usage, because even complex programs[2] load very fast, as they're mostly native executable code by virtue of the compilation approach, therefore shared by all instances of the program.

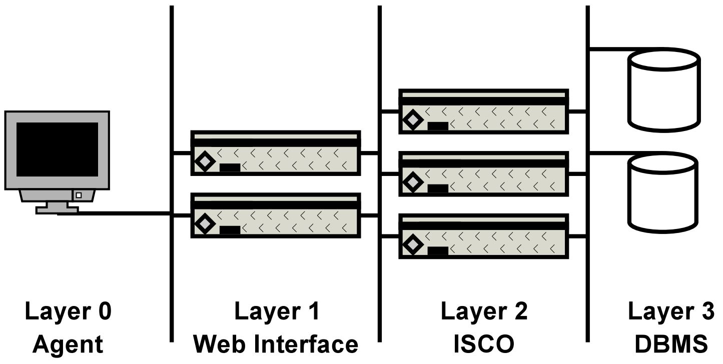

Figure 2: SIIUE.sac Physical Organizational

ISTO programs may access relational data through ODBC using a GNU Prolog interface with unixODBC, which has been developed within the SIIUE project: this allows for accessing legacy data transparently. The executables are used from within a (very thin) PHP wrapper script in web-based interfaces: the PHP extensions have also been developed specifically for use with ISTO.

Although most of the relational databases used are currently built in PostgreSQL, other relational database engines where considered. This requires ISTO to be independent from the

[2]In this case, a typical SIIUE.sac user interface program has around a hundred thousand lines of code.

specific RDBMS engine being used. The ISTO compiler is aware of the differences between relational database engines, and generates SQL code appropriate to the specific back-end being used, through the use of different units for each known database back-end, building on similarities between some to exploit multi-level specialization schemes provided by Contextual Logic Programming, as for example in dealing with different versions of the same RDBMS engine.

We fully integrated ISTO with the PiLLoW [5] library, a Prolog library for HTML/XML/SGML output and form handling, which is used for web-based development. PiLLoW has been ported to GNU Prolog.

4.2 SIIUE.sac from a Software Engineering Perspective

Universidade de Évora's commitment to develop the SIIUE.sac project was ascertained in early 2003 and the project itself got under way in March 2003 with a team of three experienced programmers. At that time, a complete rewrite of the ISTO tools and GNU Prolog/CX had just recently been rendered operational and the development team had no experience with either Contextual or Constraint Logic Programming, although they had done a few toy projects with Prolog. The project was then scheduled with quarterly milestones which targeted roughly:

1. The academic services internal use (e.g. graduation plans),
2. The student's use (e.g. course registrations) and
3. The faculty members' use (e.g. grading)

It was important that the schedule be met because this endeavour was considered mission-critical, as it involved unknowns at various levels: the technology and tools were very new and the development team was not familiar with the approach.

At the time of this writing, all of the project phases had completed successfully and were in production, having been stress-tested with both the introduction of around 40 different graduation plans ranging from Visual Arts to Veterinary Medicine, some with very intricate structures, and the registration for individual courses by approximately 6000 students, averaging 10 courses per student. Anecdotically, the most stressful moment was the first day of the student registration period, the load peak resulted in a minor problem which was resolved in under one hour.

The experience we drew from the deployment of this first application can be summed up in a few points:

- The re-use, whenever appropriate, of existing well-established software components such as Apache, PHP, PiLLoW and LaTeX was essential as it saved us a lot of specification and implementation effort.
- Contextual Logic Programming played a key role in the overall incremental design and implementation process; a few aspects deserve explicit mention:
  - The representation of user sessions as contexts was a significant success, as the concept of session can very naturally be expressed as a context.
  - Role-based authorization and interface generation gained plenty of flexibility and reliability from the systematic use of contexts.
  - Coding the "business logic" as units that respond to standardized messages (e.g. the `item/1` predicate) enabled us to design compositionally and made it relatively easy to rework implementations and restructure processes while preserving an unchanging interface.
- The choice of relegating the RDBMS to the role of persistency provider for ISTO appears to have been the correct one. This became particularly obvious at one stage, where an "SQL-like" design (structures represented as a collection of tuples or facts) was replaced with a more "Prolog-like" one (structures represented as a single large term): performance on a particular benchmark went up by a factor of 10 to 100 with that single change, the gain is mostly attributable to reduced database traffic.
- The relative ease with which programmers used to procedural languages and SQL adopted a little-documented paradigm and still very experimental development tools was surprising, as they became productive very early in the development cycle.

5 Some notes on the the GNU Prolog/CX Prototype

In order to experiment programming with contexts we have developed a first prototype inside GNU Prolog [7]. Our main goal was to have a light implementation modifying the current system as little as possible. Due to space restrictions we only give here an overview, the interested reader can consult [3] for more details.

The main change concerns a call to a given predicate `P/N`. If there is no definition for `P/N` in the global predicate table (containing all built-in predicates and predicates not defined inside a unit) then the context must be scanned until a definition is found.

N	Time (sec)	perf. loss	CSM perf. loss
0	0.971	0.0%	0.0%
1	0.986	1.5%	10.3%
2	1.004	3.4%	20.6%
5	1.043	7.4%	51.6%
10	1.102	13.5%	n/a
20	1.235	27.2%	n/a
50	1.595	64.3%	n/a
100	2.238	130.5%	n/a

Table 2: Varying context depth

To evaluate the context implementation, we followed a methodology similar to that of Denti et al. [6]: a goal is evaluated in a context which is made up of a unit which implements the goal predicate, below a variable number of "dummy" units which serve to test the overhead introduced by the context search. We used the exact same benchmark as in [6]. The results shown in table 2 correspond to average of 10 runs on a 1GHz Pentium III running Linux. The observed relative performance is much better in GNU Prolog/CX: even in CSM's most favorable situation (the modified WAM), there is a 50% performance hit as soon as there are 5 "dummy" units in the context. Finally note that the "50% performance degradation" threshold is reached when the context comprises about 40 dummy units. This demonstrates the effective ability to extensively use *deep contexts* in actual applications, and is a *sine qua non* requirement for the practical use of such a language feature.

6 Conclusions and Future Developments

ISTO has been successfully used in the design and development of a mission-critical information system. The conclusions we can draw from the experience we've gained so far include:

- A large application such as SIIUE.sac can bring out fragilities in the implementation of the tools it uses: such was the case, for instance, with GNU Prolog/CX in which a few hitherto unnoticeable bugs became manifest (and were fixed.)

- The work-in-progress status of some of the tools, most notably GNU Prolog/CX, turned out *not* to be a serious hindrance, as the design discipline made up for the lack of features such as an effective debugger.

- GNU Prolog/CX and ISTO are well suited to o incremental OO design, as a system can become operational even while still incomplete and underspecified.

- One feature often touted as a must-have for Prolog implementations is the (heap) garbage collector: the lack of one turned out not to be a limiting factor, as Prolog processes have relatively short lifetimes: they only need to compute one individual page in a session, the continuation being served by the next process.

- Complex SQL code generation is not as important a goal as we initially thought it would, because it can largely be compensated by the judicious use of the result of simple queries.

- The gradual adoption of Contextual Logic Programming as a design and programming paradigm has exhibited not too steep a learning curve and is allowing us to further our sensitivity to its different applicability situations and program patterns. Some of these reflect back onto the language itself.

- The developers did have to rid themselves from SQL and procedural language habits, namely in what concerns the manipulation of more complex data structures, in order to extract acceptable performance from the system.

Now that the first full-size application is finished, some directions become apparent for the future development of ISTO and GNU Prolog/CX, as the scope of their use widens:

- The removal of some implementation-specific limits (e.g. area sizes) and some low-level extensions such as the dynamic loading of compiled Prolog code, which will allow for on-the-fly extension of compiled applications or multi-thread execution.

- The development of a generic web-based relation browser, as this will greatly decrease interface development time which has – once again – proven to constitute the bulk of the development effort.

- A more extensive performance analysis and tuning under load.

Acknowledgements

The work described herein was partly made possible by the bilateral INRIA/GRICES project "Extensions au Logiciel Libre GNU Prolog." Universidade de Évora is acknowledged for supporting and funding the SIIUE.sac project.

References

[1] Salvador Abreu. A Logic-based Information System. In Enrico Pontelli and Vitor Santos-Costa, editors, *2nd International Workshop on Practical Aspects of Declarative Languages (PADL'2000)*, volume 1753 of *Lecture Notes in Computer Science*, pages 141–153, Boston, MA, USA, January 2000. Springer-Verlag. 1, 3, 4

[2] Salvador Abreu. Isco: A practical language for heterogeneous information system construction. In *Proceedings of INAP'01*, Tokyo, Japan, October 2001. INAP. 1, 2.3, 3

[3] Salvador Abreu and Daniel Diaz. Objective: in Minimum Context. In Catuscia Palamidessi, editor, *Logic Programming, 19th International Conference, ICLP 2003, Mumbai, India, December 9-13, 2003, Proceedings*, volume 2916 of *Lecture Notes in Computer Science*, pages 128–147. Springer-Verlag, 2003. ISBN 3-540-20642-6. 2, 2.2, 5

[4] M. Bugliesi. A declarative view of inheritance in logic programming. In Krzysztof Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 113–127, Washington, USA, 1992. The MIT Press. 2

[5] Daniel Cabeza and Manuel Hermenegildo. Distributed WWW programming using (Ciao-)Prolog and the PiLLoW library. *Theory and Practice of Logic Programming*, 1(3):251–282, May 2001. 2

[6] Enrico Denti, Evelina Lamma, Paola Mello, Antonio Natali, and Andrea Omicini. Techniques for implementing contexts in Logic Programming. In Evelina Lamma and Paola Mello, editors, *Extensions of Logic Programming*, volume 660 of *LNAI*, pages 339–358. Springer-Verlag, 1993. 3rd International Workshop (ELP'92), 26–28 February 1992, Bologna, Italy, Proceedings. 5

[7] Daniel Diaz and Philippe Codognet. Design and implementation of the gnu prolog system. *Journal of Functional and Logic Programming*, 2001(6), October 2001. 5

[8] Francis G. McCabe. *Logic and Objects*. Prentice Hall, 1992. 2

[9] Dale Miller. A logical analysis of modules in logic programming. *The Journal of Logic Programming*, 6(1 and 2):79–108, January/March 1989. 2

[10] Luís Monteiro and António Porto. A Language for Contextual Logic Programming. In K.R. Apt, J.W. de Bakker, and J.J.M.M. Rutten, editors, *Logic Programming Languages: Constraints, Functions and Objects*, pages 115–147. MIT Press, 1993. 2

[11] Antonio Natali and Andrea Omicini. Objects with State in Contextual Logic Programming. In Maurice Bruynooghe and Jaan Penjam, editors, *Programming Language Implementation and Logic Programming*, volume 714 of *LNCS*, pages 220–234. Springer-Verlag, 1993. 5th International Symposium (PLILP'93), 25–27 August 1993, Tallinn, Estonia, Proceedings. 2

[12] Vitor Beires Nogueira, Salvador Abreu, and Gabriel David. Towards Temporal Reasoning in ISCO. In *Proceedings of AGP'02*, Madrid, Spain, 2002. FI/UPM. 3.2

[13] Vitor Beires Nogueira, Salvador Abreu, and Gabriel David. Using Contextual Logic Programming for Temporal Reasoning. In Ernesto Pimentel and Nieves R. Brisaboa, editors, *VIII Conference on Software Engineering and Databases (JISBD 2003)*, Alicante, Spain, November 2003. 3.2

[14] António Porto. An Integrated Information System Powered by Prolog. In Verónica Dahl and Philip Wadler, editors, *Practical Aspects of Declarative Languages, 5th International Symposium, PADL 2003, New Orleans, LA, USA, January 13-14, 2003, Proceedings*, volume 2562 of *Lecture Notes in Computer Science*, pages 92–109. Springer, 2003. 1

[15] David Ungar and Randall B. Smith. Self: The Power of Simplicity. In Norman K. Meyrowitz, editor, *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87), October 4-8, 1987, Orlando, Florida, Proceedings*, volume 22 of *SIGPLAN Notices*, pages 227–242, December 1987. 2.1