Paving the Roadmaps: Enabling and Integration Technologies

Danny De Schreye, Manuel Hermenegildo and Luís Moniz Pereira

Katholieke Universiteit Leuven,
Technical University of Madrid,
Universidade Nova de Lisboa.
email: dannyd@cs.kuleuven.ac.be herme@fi.upm.es
lmp@di.fct.unl.pt

1 Introduction

In this chapter of the roadmap, we first discuss challenges for two specific supportive enabling technologies: constraint programming and systematic program development. These two areas have been prominent in Computational Logic research and applications for over a decade. We briefly assess achievements and point out challenging future works in these areas.

Next, we present a broader view on Enabling and Integration technologies (EIT). Here, we discuss issues that are not specifically concerned with any particular Computational Logic topic, but with the global concerns applicable to each and every topic. And in particular those issues raised in other topical roadmap chapters where reference to the EIT concerns is made.

Indeed, keeping to the analogy, the EITs provide necessary and common paving support for efficient and reliable road construction and use. We introduce four EITs dimensions which help organize the issues involved: Investment, Generality, Synthesis, and Systems, and comment on each of these.

2 Some Challenges for Constraint Programming

2.1 Introduction

Constraint programming languages [27, 20] are among the very few new language proposals that are seeing industrial success while being based on a novel programming paradigm. Intervening factors in this fact appear to be

the expressiveness awarded by the high level of these languages, the combination of search and incremental constraint solving capabilities, and the relative efficiency of the resulting applications. These characteristics are added to the strengths in general-purpose symbolic processing of the underlying logic programming kernel on which they are often built. Constraints extend this kernel to numerical domains and beyond, offering a natural platform in which applications combining symbolic and numeric computing can be easily developed. This initial success notwithstanding, it seems important to identify significant challenges which lay ahead for constraint programming. Without any pretense of being exhaustive, we herein touch upon some challenges in the areas of implementation technology, network-wide programming, and language and system design.

2.2 Faster performance through global analysis

The performance of current constraint logic programming systems often compares favorably to other constraint solving and symbolic programming tools. However, their performance is still lower than that of traditional imperative languages, specially in (non constraint related) numerical computations. A possible solution is to develop advanced compilation technology capable of detecting the cases where limited or no constraint solving is involved and compiling those cases in the most efficient way. Some promising progress has already been made towards this goal. The efficiency and effectiveness of global analysis of logic programs is already established [46], and essentially the same performance as imperative languages has been achieved in some experimental systems (e.g., [39]). Some results on the possible speedups attainable in constraint logic programming have been reported (e.g., [36, 15]), some practical frameworks for global analysis developed (e.g., [10]), and a few systems described which perform global analysis based optimization [29, 15]. Such global analysis has also been applied to concurrent constraint programming systems, where one of the most important objectives is to reduce suspension and resumption of goals and synchronization overhead [9, 33, 34, 30, 35, 17]. Recent progress in incremental and compositional global analysis [33, 23, 4] appears to solve most remaining problems related to dealing with large programs and the interactive program development environment that is common in constraint programming systems. However, the application of extensive optimization in commercial or widely used public domain systems still remains a goal to be achieved. Also, much research remains to be done in finding accurate abstraction domains for standard constraint systems.

2.3 Faster performance through automatic parallelization

Program parallelization is becoming a more and more interesting optimization since multiprocessing hardware is starting to be in many cases the default installation platform. This is often the case, for example, for departmental servers where multiprocessors using using fast, inexpensive, off-the-shelf processors are replacing mainframes at a fraction of their cost. Some high-end workstations are also multiprocessors. It appears likely that this trend towards increased use of parallelism will continue as multiprocessor architectures are better understood, interconnection network performance increases with new technologies (specially if the promise of optical interconnect is finally delivered), and feature size diminishes allowing placement of several processors on the same chip.

The interest of the study of parallelism in logic and constraint logic programs goes beyond the paradigms themselves. In fact, these languages allow studying challenging issues in parallelization while remaining in the context of a relatively clean and well understood framework for which application codes are available. Such challenging issues include inter-procedural parallelization, irregular computations, recursion, complex data structures with (well behaved) pointers, and speculative parallelism. These issues have already been studied in the context of logic programs [6], where significant speedups have been obtained. In the context of constraint logic programming, exploitation of parallelism in the search (or-parallelism) is comparatively easy and has been shown to achieve significant speedups in constraint programs containing independent search [21, 14, 13]. On the other hand, and as pointed out in [27], comparatively little work has been devoted so far to exploiting parallelism within a given path of the search (and-parallelism) and in the solver itself. Although traditional concepts of independence used in imperative programming (e.g., the "Bernstein conditions") or even those of logic programming, do not apply in the context of CP [16], notions of independence appropriate for (concurrent) CP have been recently proposed [16, 5]. UPM, Madrid, has since been tackling the issue of automatic parallelization and has recently developed an automatic parallelizing compiler for constraint logic programs, with encouraging initial results [15]. While it may prove the case that there is often not much parallelism in constraint solving kernels, we believe that over the set of all programs the technology will eventually prove as successful as in the context of logic programming. However, proving this and developing new parallelization technology for constraint logic programs remains an area for future work.

2.4 Improved debugging and visualization facilities

An area in which current constraint logic programming systems have been found lacking by users developing industrial applications is debugging, in terms of ensuring correctness as well as understanding and improving performance of the program. Possible solutions can be based on both static and dynamic techniques. One technique is the use of assertions [12, 4]. Assertions can be seen as a generalization of type systems (often including directional types), in which relatively general preconditions and postconditions can be declared for procedures. Assertions can be provided by the user and checked at compile-time or run-time. Compile-time checking can in some cases be done via global analysis. Furthermore, assertions can be generated by the compiler, which the user can then inspect in order to detect high-level errors (here, the same technology used by the compiler for optimization purposes can be used). Another possible technique is the use of visualization, both of the search space and of the constraint store at different points of the execution (e.g., [32]). The development of more useful assertions, assertion checking and generation algorithms, and visualization paradigms for constraint programming remains a challenge.

2.5 Distributed and network-wide programming

Another challenge for constraint programming systems is related to the role of such systems in network-wide programming. This type of programming is likely to be of growing importance since the recent wider diffusion of the Internet and the popularity of the "World Wide Web" –WWW– protocols, which effectively provide a platform that is standard and ubiquitous and allows a new class of highly sophisticated distributed applications. Constraint logic programming systems already offer many characteristics which appear to place them well for making an impact in this area. Some of these characteristics, including dynamic memory management, well behaved structure and pointer manipulation, robustness, and compilation to architecture independent byte-code, are shared with other proposed network programming tools. In addition, constraint logic programming systems offer a quite unique set of other built-in features including dynamic databases, search facilities, grammars, sophisticated meta-programming, constraint solving capabilities, and well understood semantics. In addition, the theory of concurrent constraint programming offers a novel and very rich notion of inter-process communication based on constraint entailment [31, 43]. Some of these characteristics, such as the built-in database and grammars, can facilitate the

rapid coding of relatively simple applications, even by naive end users. Furthermore, it seems natural that the more sophisticated network applications, such as intelligent information retrievers or distributed decision making systems, will require complex symbolic and numeric capabilities, including natural language processing, at which constraint programming (and, in particular, constraint logic programming) has already been shown particularly successful [37].

Sophisticated distributed applications can be developed with current constraint logic programming systems using the available low-level primitives to build communication abstractions such as blackboards [1, 7, 44] and even incorporating distributed variable-based communication (which can be supported, for example, using attributed variables [19]). In our experience the main feature found lacking in standard systems is native support for concurrent execution. While many systems allow starting a whole subprocess via the operating system interface and this can provide the desired functionality, it is very inefficient. A more direct support for concurrency (perhaps along the lines of that present in the AKL [26], Oz [42], and CIAO [24, 18] systems) seems highly desirable and we feel should be provided in all future constraint programming systems. One additional important use of concurrency is in implementing complex, delay based constraint solving algorithms. However, concurrency brings important new challenges in many areas. As mentioned before, an important one from the implementation point of view is developing effective analysis and optimization techniques.

Distributed concurrent constraint systems are currently being worked on [7, 25], distribution and application development libraries are being offered (e.g., [19, 8]), and network and WWW applications are being reported [45]. CP is a promising foundation for many aspects of the next generation of distributed systems, but it still remains as a challenge to develop simple, elegant and practically usable environments, and to demonstrate applications of such environments.

2.6 The CIAO system

One particular concurrent constraint logic programming system, "CIAO", illustrates how UPM in Madrid is approaching some of the challenges that we have set forward in the previous paragraphs [24, 18]. At the user level CIAO is a programming environment offering support for full standard Prolog, as well as several constraint domains, several control rules, a module and object system, and concurrency, synchronization, and distribution primitives. As opposed to other concurrent constraint languages such as AKL [26] or

OZ [42], the language is "sequential by default", in the sense that concurrency is annotated explicitly by the user. Optional control rules include for example the determinate-first principle [41]. A rich set of assertions is available which the compiler understands and both generates and checks by means of global analysis techniques based on abstract interpretation. These assertions include a flexible type system.

At the implementation level most functionality is implemented via source to source transformation into a simple kernel language, which is then the subject of extensive analysis and optimization, using a single set of analysis and transformation tools. High-level optimizations include parallelization, granularity control, reduction of concurrency and synchronization, reordering of goals, code simplification, and specialization. Distributed and network-wide programming are implemented via libraries [19, 8]. One of the main features of the kernel language and the abstract machine underlying it is the support for concurrency and parallel execution, based on an extension of the capabilities of the &-Prolog engine [22]. CIAO is currently being used both as a workbench for testing implementation techniques (many of the program analysis and transformation systems referenced elsewhere have been prototyped on CIAO) and a powerful programming environment. Current versions of the &-Prolog and CIAO systems, and the related libraries are publicly available for experimentation (see `http://www.clip.dia.fi.upm.es/` for both software and publications).

3 Systematic development of programs and systems

3.1 Introduction

Systematic techniques for analyzing, optimizing and developing programs and systems have formed one of the key areas of research in Computational Logic for several decades. There have been two main reasons for the strong research activity in this area. The first is that in Computational Logic, unlike in some other paradigms, programs have a clear and often simple formal semantics. Having such a clear formal semantics makes it easier to develop tools that analyze and optimize programs. In particular, it ensures that reasoning about the correctness of such analysis and optimization tools is much facilitated. Secondly, but closely related to the above, the logic and the control of programs written in Computational Logic languages can most often be considered as separate, orthogonal components of the program. For

program optimization in particular, this is an open invitation to tune the control component towards the problem at hand, while leaving the logic invariant.

3.2 Critical assessment of the results

The work in this area has produced many breakthroughs. Very refined analysis and specialization tools have been developed. Witnesses of the success are the many related surveys in [3] and, more recently, the collected papers in [2], but also the numerous papers in this area that appeared in the major conferences and workshops ICLP, ILPS, PLILP, SAS, LOPSTR and META over the years.

However, one could argue that the success is only partial. In particular, the following observations come to mind:

- Very few of the developed techniques have been integrated in development environments of real commercial systems.
- Many contributions, in particular in program synthesis and transformation, have only been applied to programming in the small. It is insufficiently clear whether they scale up. Nor is it clear whether the techniques can be made sufficiently efficient to practically deal with larger programs.
- Furthermore, it is unclear how these techniques should be integrated in the software engineering process and how they can be combined with software engineering methodology in general.
- Finally, much of the work has been restricted to a specific execution strategy for Logic Programs, the so called left-to-right SLD-resolution of Prolog. This is somewhat outdated and results should be upgraded to deal with the many more recently introduced execution mechanisms, like those present in Constraint Logic Programming solvers, abductive solvers ([28]) and solvers using tabling [47].

3.3 A high level road map for the Development area

There are various reasons for the partial lack of success with respect to the issues above. One overall reason is that the problems tackled in this area are very hard. Advanced semantics-based program optimization comes close to achieving the goals of automated programming. Considering that these goals are very ambitious and hard, it is not surprising that the progress

made in this area is relatively slow. Moreover, due to the introduction of many new solvers in novel Computational Logic languages, much of this work needs revised in many different new directions.

Returning on the issues mentioned in the previous section, integration of these tools in development environments of practical programming systems was not well achieved. In part, this has been related to the relatively weak commercial position of the vendors. This did not allow them to invest in integrating research prototypes into their systems. Also, a main bottleneck has been the absence of any commercial-quality public domain language. Having such a language would have allowed the research groups to attach their prototypes to this language, making them available to a wider community for experimentation and possible integration with other such tools.

Recently, the situation has considerably improved. The language Mercury [38] is providing a testing ground for several new techniques. Also, the work of P.Codognet has led to the development of a 'gnu-Prolog'. It is one of the important tasks within this area to connect newly developed tools and techniques to these systems, so that they become available for experimentation and for integration and cross-fertilization with other techniques.

With respect to scaling up the techniques to be able to cope with more than just programming in the small, several issues arise. Concerning mere 'size' of programs, many of the developed techniques have insufficiently considered the performance-precision trade-off. More work needs to be invested in studying to what extent an acceptable precision of the analysis and optimization tools can be combined with high run-time performance. Some work in this direction have already been done and lead to very promising results. For instance, again in the context of Mercury, a very efficient termination analyzer was able to successfully analyze very large programs, including the Mercury compiler ([40]).

A different dimension in scaling up techniques involves the integration of the analysis and optimization techniques with software development methods for programming in the large. An important issue that needs to be investigated here is compositionality of the techniques. To some extent, the closed world assumption, inherent in standard Logic Programming, forms a considerable difficulty with respect to compositionality. Open or abductive Logic Programs [28] provide a much better setting to support it.

But even more important than compositionality is the study of the role of development techniques within state of the art software engineering methodologies. Computational Logic does not have any direct counterpart for object oriented software analysis and design. Such counterpart must be developed and the role of automated development tools within it must be clarified.

Finally, the potential role of these tools within component-based software development should be investigated.

Probably the most important challenge on the road map for the program development area lies in scaling up the techniques to allow them to cope with richer solvers than SLD(NF)-resolution. Over the last decade, very many new solvers for Computational Logic have been developed. Even when restricted to Constraint Logic Programming (CLP), there is a huge variety of proposed methods to replace and/or refine and extend SLD(NF)-resolution. Typical examples are hybrid techniques based on backtracking combined with arc-consistency techniques, such as forward checking, looking ahead and partial looking ahead. But also methods from Operations Research, for instance, those based on the simplex method, methods based on interval arithmetic, and many others are available in CLP-languages. Moreover, outside the CLP-area, solvers that integrate abductive reasoning or tabling in SLD(NF)-resolution have become popular. In the near future, one may also expect integrations of tabling with CLP and/or abduction.

In view of this high diversity of different new solvers, developing specific analysis or optimization tools for each of these seems infeasible. A more realistic goal is to select the most frequently used practical solvers and to develop specialized tools only for these. Another direction is to produce generic techniques that are applicable to a family of different, but similar solvers.

Yet another promising line of future work is to develop tools to transform models of problem domains. One of the important problems in CLP is that, given a problem domain, it is often difficult to select the appropriate model for that domain. Which concepts in the problem domain should be represented by variables and which ones by value sets? In many problems, the role of the variables and the value sets can be interchanged to obtain a new model. Also, it is often unclear whether one should use a boolean representation or a finite domain one. Even further, it if often unclear whether one would better represent a problem in CLP or in an abductive Logic Program.

It would be interesting if these different representations could be automatically transformed from one into the other. This would allow to experiment quickly with various alternative models and solvers for a same problem and select the most promising one.

In conclusion, although program analysis and optimization techniques have been studied for a long time in Computational Logic, we believe that there are still many promising challenges for that area in the context of new emerging technologies. The advantages that Computational Logic of-

fers with respect to achieving successes in automated program development support need to be further exploited. We refer to [11] for a more elaborated presentation of most of the issues in this section.

4 A Broader Assessment of Enabling and Integration Technologies in Computational Logic

Here we move away from concrete enabling technologies and introduce four EITs dimensions which help organize the issues involved: Investment, Generality, Synthesis, and Systems.

4.1 Investment

A wealth of investment in Computational Logic research has accumulated in the form of published results and built systems, over the more than 25 years since its inception. Some of these could not be brought to full fruition at the time because of technological costs and funding limitations, or more simply on because of the lack in availability of cheap and efficient requisite equipment, nowadays having become affordable and performant. So, though the theory and implementational know-how were there at the time, their full fruition was not viable. However, to-day, younger researchers are not well aware of such past results and of their potential, some from the early eighties, even when the implementational technology may have evolved to allow them to be put to use.

Illustrative case in point examples are those of "intelligent backtracking" and of "declarative debugging". And all the more so because the complexity of present systems makes such immediately available "theory technologies" all the more desirable. On the other hand, the cost of memory and the appearance on the scene of new "implementation technologies", such as (theory-driven) tabling, make the prior investment in theory not only realizable now but affordable and profitable as well.

Conversely, we should not stop or refrain from investing now for the future, for the "theory technologies" of today, in addressing real felt problems, will be the providers of tomorrow's implementational and applicational reapers.

For instance, were it not for the impressive development in Logic Programming semantics in the past 12 years or so, and its extensions to nonmonotonic and other forms of reasoning, we would not have today the impressive, sound, and efficient system implementations of such semantics (be

it the stable or well-founded varieties or their hybrid combination), which have been opening up a whole new gamut of application areas as well as a spate of sophisticated reasoning abilities over them. This was made possible only through successive and prolonged efforts at theoretical generalization and synthesis, and by way of the combined integration of theory, procedure development, and practical implementation.

The lesson, then, for further investing in the EITs is that it must be continual, both for the fruition of past investment as for the seeding of timely future delectation.

4.2 Generality

The Computational Logic paradigm provides a well-defined, general, integrative, encompassing, and rigorous framework for systematically studying computation, be it syntax, semantics, procedures, or attending implementations, environments, tools, and standards. Computational Logic approaches problems, and provides solutions, at a sufficient level of abstraction so that they generalize from problem domain to problem domain. This is afforded by nature of its very foundation in logic, both in substance and method, and constitutes one of its major assets.

Indeed, computational reasoning abilities such as assuming by default, abducing, revising beliefs, removing contradictions, updating, planning, learning, constraint handling, etc etc, by dint of their generality and abstract characterization, once developed can readily be adopted by, and integrated into, distinct topical application areas.

Consequently, it pays enormously to promote such basic EITs. Conversely, topical areas will have their development hindered if such pervasive EITs are not continually fostered.

4.3 Synthesis

Computational Logic is the privileged melting pot for the articulate integration of functionalities and techniques, pertaining to the design and mechanization of complex systems, addressing ever more demanding and sophisticated computational abilities.

For instance, consider again those reasoning abilities just mentioned in the previous section. Forthcoming rational agents, to be realistic, will require an admixture of any number of them to carrying out their tasks. No other computational paradigm affords us with the wherewithall for their coherent conceptual integration. And, all the while, the very vehicle that

enables testing its specification, when not outright its very implementation. Or is there?

Moreover, the issue is not just that of conceptual integration but that of conceptual cross-fertilization too. How can one employ learning in the service of planning? How does one remove contradictions from updates? How can learnt preferences be combined with preference updates to guide planning in a rational agent? How may fuzzy preferences be revised in the light of ongoing abduced assumptions and constraints?

The groundwork for the EITs required to sustain the principled treatment with provable properties of such foreseeable and inescapable issues (many are already upon us) is part and parcel of the roadmap paving activity and maintenance, without which roads will subside, fail in their reliability, or might not be enlargeable to serve newly developed needs.

4.4 Systems

Without systems (engines, environments, integration and development tools) we have no vehicles for actually traveling on the roads. No roadmap is complete without them, as no map can be an end in itself.

Much progress has been achieved of late in incorporating into systems not just the theoretical developments in languages, semantics, and procedures, but also the improvements in design and implementation technology, which partake of innovations in equipment and networks.

However, more effort and support is needed in the way of improved descriptional abstraction of innovative solutions, and thus of their improved transmigration among systems. Also, system hybridization is an issue in itself, so that the best features and functionalities of different systems may become more articulate-able, and brought to bear on complex problems demanding the specialized handling of distinct aspects of a problem. Promoting common standards will be of help here of course.

In short, ever better and more complete systems constitute an EITs goal we cannot do without, at the risk of foregoing the whole Computational Logic enterprise, and making its roads uninhabited and uninhabitable.

4.5 Conclusion

Supporting the Enabling and Integrative Technologies, both theoretical and implementational, is an inescapable joint priority for making viable, in a sustained way, the diverse topical priorities in Computational Logic. So be it!

References

[1] J. Almgren, S. Andersson, L. Flood, C. Frisk, H. Nilsson, and J. Sundberg. *Sicstus Prolog Library Manual.* Po Box 1263, S-16313 Spanga, Sweden, October 1991.

[2] A. Bossi and Y. Deville, editors. The Journal of Logic Programming, Special Issue on Synthesis, Transformation and Analysis. Vol. 39/41, Elsevier Science Publishers, 1999.

[3] M. Bruynooghe, S. Debray, M. Hermenegildo and M. Maher, editors. The Journal of Logic Programming, Tenth Anniversary Issue. Vol. 19/20, Elsevier Science Publishers, 1994.

[4] F. Bueno, D. Cabeza, M. Hermenegildo, and G. Puebla. Data–flow Analysis of Standard Prolog Programs. In *European Symposium on Programming*, number 1058 in LNCS, pages 108–124, Sweden, April 1996. Springer-Verlag.

[5] F. Bueno, M. Hermenegildo, U. Montanari, and F. Rossi. From Eventual to Atomic and Locally Atomic CC Programs: A Concurrent Semantics. In *Fourth International Conference on Algebraic and Logic Programming*, number 850 in LNCS, pages 114–132. Springer-Verlag, September 1994.

[6] J. Chassin and P. Codognet. Parallel Logic Programming Systems. *Computing Surveys*, 26(3):295–336, September 1994.

[7] 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.clip.dia.fi.upm.es/`.

[8] D. Cabeza, M. Hermenegildo, and S. Varma. The PiLLoW/CIAO Library for INTERNET/WWW Programming using Computational Logic Systems. In *Proceedings of the 1st Workshop on Logic Programming Tools for INTERNET Applications*, JICSLP"96, Bonn, September 1996. Text and code available from `http://www.clip.dia.fi.upm.es/miscdocs/pillow/pillow.html`.

[9] M. Codish, K. Marriott M. Falaschi, and W. Winsborough. Efficient analysis of concurrent constraint logic programs. In *Twentieth International Coll. Automata, Languages and Programming*, Lund, Sweden, July 1993.

[10] M. García de la Banda, M. Hermenegildo, M. Bruynooghe, V. Dumortier, G. Janssens, and W. Simoens. Global Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems*, 18(5):564–615, 1996.

[11] D. De Schreye and M. Denecker. Assessment of some issues in cl-theory and program development. In K. Apt, V. Marek, M. Truszczynski, and D. Warren, editors, *The Logic Programming Paradigm: a 25 Years Perspective*. Springer-Verlag, *Artificial Intelligence Series*, 1999.

[12] W. Drabent, S. Nadjm-Tehrani, and J. Maluszynski. Algorithmic debugging with assertions. In H.Abramson and M.H.Rogers, editors, *Meta-programming in Logic Programming*. MIT Press, 1989.

[13] L. Li et al. APPLAUSE: Applications using the ElipSys parallel CLP system. In *Proceedings of the International Conference on Logic Programming*, pages 847–848. MIT Press, 1993.

[14] European Computer Research Center. *Eclipse User's Guide*, 1993.

[15] M. García de la Banda, F. Bueno, and M. Hermenegildo. Towards independent And-Parallelism in CLP. In *International Symposium on Programming Language Implementation and Logic Programming, PLILP'96*, volume 1140 of *LNCS*, pages 77–91. Springer Verlag, September 1996.

[16] M. García de la Banda, M. Hermenegildo, and K. Marriott. Independence in Constraint Logic Programs. In *1993 International Logic Programming Symposium*, pages 130–146. MIT Press, Cambridge, MA, October 1993.

[17] M. García de la Banda, K. Marriott, and P. Stuckey. Efficient Analysis of Constraint Logic Programs with Dynamic

Scheduling. In *1995 International Logic Programming Symposium*, pages 417–431, Portland, Oregon, December 1995. MIT Press, Cambridge, MA.

[18] M. Hermenegildo, F. Bueno, M. García de la Banda, and G. Puebla. The CIAO Multi-Dialect Compiler and System: An Experimentation Workbench for Future (C)LP Systems. In *Proceedings of the ILPS'95 Workshop on Visions for the Future of Logic Programming*, Portland, Oregon, USA, December 1995.

[19] M. Hermenegildo, D. Cabeza, and M. Carro. Using Attributed Variables in the Implementation of Concurrent and Parallel Logic Programming Systems. In *Proc. of the Twelfth International Conference on Logic Programming*, pages 631–645. MIT Press, June 1995.

[20] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.

[21] P. Van Hentenryck. Parallel Constraint Satisfaction in Logic Programming. In *Sixth International Conference on Logic Programming*, pages 165–180, Lisbon, Portugal, June 1989. MIT Press.

[22] M. Hermenegildo and K. Greene. The &-Prolog System: Exploiting Independent And-Parallelism. *New Generation Computing*, 9(3,4):233–257, 1991.

[23] M. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental Analysis of Logic Programs. In *International Conference on Logic Programming*, pages 797–811. MIT Press, June 1995.

[24] 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*, LNCS 874, pages 123–133. Springer-Verlag, May 1994.

[25] S. Haridi, P. VanRoy, and G. Smolka. An Overview of the Design of Distributed Oz. In *Proc. of the 1996 JISCLP Workshop*

on Multi-Paradigm Logic Programming. T.U.Berlin, September 1996.

[26] S. Janson and S. Haridi. Programming Paradigms of the Andorra Kernel Language. In *1991 International Logic Programming Symposium*, pages 167–183. MIT Press, 1991.

[27] J. Jaffar and M.J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 13/20:503–581, 1994.

[28] A. C. Kakas, R. Kowalski and F. Toni. The Role of Abduction in Logic Programming. Handbook of Logic in Artificial Intelligence and Logic Programming 5. Oxford University Press, pages 235–324, 1998.

[29] A. Kelly, A. Macdonald, K. Marriott, P. Stuckey, and R. Yap. Effectiveness of optimizing compilation for CLP(R). In *Proceedings of Joint International Conference and Symposium on Logic Programming*, pages 37–51. MIT Press, 1996.

[30] Andy King and Paul Soper. Schedule Analysis of Concurrent Logic Programs. In Krzysztof Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 478–492, Washington, USA, 1992. The MIT Press.

[31] M. J. Maher. Logic Semantics for a Class of Committed-choice Programs. In Jean-Louis Lassez, editor, *Proceedings of the Fourth International Conference on Logic Programming*, Series in Logic Programming, pages 858–876, Melbourne, 1987. MIT Press.

[32] M. Meier. Grace User Manual, 1996. Available at `http://www.ecrc.de/eclipse/html/grace/grace.html`.

[33] K. Marriott M. Falaschi, M. Gabbrielli and C. Palamidessi. Compositional analysis for concurrent constraint programming. In *IEEE Symposium on Logic in Computer Science (LICS)*, Montreal, June 1993.

[34] K. Marriott, M. García de la Banda, and M. Hermenegildo. Analyzing Logic Programs with Dynamic Scheduling. In *20th. Annual ACM Conf. on Principles of Programming Languages*, pages 240–254. ACM, January 1994.

[35] U. Montanari, F. Rossi, F. Bueno, M. García de la Banda, and M. Hermenegildo. Towards a Concurrent Semantics based Analysis of CC and CLP. In *Principles and Practice of Constraint Programming*, LNCS 874, pages 151–161. Springer-Verlag, May 1994.

[36] K. Marriott and P. Stuckey. The 3 R's of Optimizing Constraint Logic Programs: Refinement, Removal, and Reordering. In *19th. Annual ACM Conf. on Principles of Programming Languages*. ACM, 1992.

[37] The practical application of constraint technology conference series. The Practical Application Company, 54 Knowle Avenue, Blackpool, Lancs FY2 9UD, U.K.

[38] Z. Somogyi, F. Henderson and T. Conway. The execution algorithm of Mercury, an efficient purely declarative Logic Programming language, The Journal of Logic Programming, 1996, 29, pages 17–64.

[39] P. Van Roy and A.M. Despain. High-Performace Logic Programming with the Aquarius Prolog Compiler. *IEEE Computer Magazine*, pages 54–68, January 1992.

[40] C. Speirs, Z. Somogyi and H. Sondergaard Termination Analysis for Mercury. In *Proceedings of the Fourth International Symposium on Static Analysis*, P. Van Henteryck, Ed. Number 1302 in LNCS. Springer, Paris, France, 157–171, 1997.

[41] V. Santos-Costa, D.H.D. Warren, and R. Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-parallelism. In *Proceedings of the 3rd. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, April 1990.

[42] Gert Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Computer Science Today*, Lecture Notes in Computer Science, vol. 1000, pages 324–343. Springer-Verlag, Berlin, 1995.

[43] V. Saraswat, M. Rinard, and P. Panangaden. Semantic Foundation of Concurrent Constraint Programming. In *Proceedings*

of the 18th. Annual ACM Conf. on Principles of Programming Languages. ACM, 1991.

[44] Paul Tarau and Koen De Bosschere. Virtual World Brokerage with BinProlog and Netscape. In *Proceedings of the 1st Workshop on Logic Programming Tools for INTERNET Applications*, JICSLP"96, Bonn, September 1996.

[45] Paul Tarau, Andrew Davison, Koen De Bosschere, and Manuel Hermenegildo, editors. *Proceedings of the 1st Workshop on Logic Programming Tools for INTERNET Applications*, JICSLP"96, Bonn, September 1996.

[46] R. Warren, M. Hermenegildo, and S. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *Fifth International Conference and Symposium on Logic Programming*, pages 684–699, Seattle, Washington, August 1988. MIT Press.

[47] D. S. Warren. Memoing for Logic Programs. CACM, Vol. 35, 3, pages 93–111, 1992.