On Benchmarking Constraint Logic Programming Platforms

Response to Fernandez and Hill's "A Comparative Study of Eight Constraint Programming Languages Over the Boolean and Finite Domains"

Mark Wallace, Joachim Schimpf, Kish Shen and Warwick Harvey[*]

November 8, 2001

1 Introduction

1.1 Background

Benchmarking is a difficult business. This is already true for benchmarking hardware and C compilers, but is even more so for complex constraint programming systems. The challenge is to establish clear objective measures of performance, and to set up tests that compare like with like.

The comparative study published in this journal by Fernandez and Hill benchmarked some constraint programming systems on a set of well-known puzzles. In this paper we shall examine the positive and negative aspects of this kind of benchmarking.

As members of the ECL^{i}PS^e [WNS97] development team, we were involved in a rather different benchmarking exercise, comparing constraint programming platforms in the context of the CHIC-2 ESPRIT project. We will discuss a number of relevant aspects of this work, and in conclusion will review the lessons learned from these two approaches to tackling the thorny issue of comparing CLP systems.

1.2 What is being Benchmarked?

Theoretically, the ideal way to choose the best CLP system for an application is to implement a solution for the application with each system and choose the best. In practice, however, one can only benchmark the systems on a predefined set of problems, and hope that the results on the benchmark problem set can be extrapolated to the required application.

[*]The authors belong to the ECL^{i}PS^e team at IC-Parc.

For a particular application one should choose the best CLP system according to expressiveness and efficiency:

1. Good expressiveness enables a system to encode a problem in terms of a short, easy-to-understand program.
2. Efficiency has two aspects:
   (a) the time needed to develop a program to solve a problem and
   (b) the computing resources needed to execute it.

   Naturally with more time spent on development, it is often possible to write a program which executes faster. One drawback of benchmarking CLP systems on applications is that most such systems can be tailored to a particular application by hard-wiring certain constraints, or by dropping into a lower-level language. Consequently the performance of the "most efficient" program may have little to do with the CLP system itself.

Some de facto standards have emerged for some subsets of CLP. In particular finite domain solvers typically support a standard set of constraints. These standard constraints can be directly benchmarked against each other by running them in isolation, which is termed *unit testing*. Aspects of CLP systems that can be measured by unit testing are:

- computation time,
- memory consumption and
- scalability.

In cases where different constraints interact, they can also be unit tested in specific combinations.

Unfortunately there will typically also be interactions with other functionalities of the CLP system, which may not be standard. For example finite domain constraints may interact with:

- intelligent backtracking and advanced search techniques;
- extensibility and flexibility, such as a facility for the programmer to dictate the priorities of different constraints;
- continuous variables, interval and linear constraints; and
- debugging, graceful error handling and explanation facilities.

In summary, a benchmarking exercise should cover a broad set of representative problems and a broad set of programming constructs. This can be achieved using two kinds of benchmarking:

1. **Applications Benchmarking** - by using the different systems to solve a predefined set of problems.

2. **Unit Testing** - by timing and measuring individual features of the different systems.

However there are risks in both these ways of benchmarking CLP systems:

1. for *applications benchmarking*, because the most efficient solution may well be achieved by writing specialised low-level code; and
2. for *unit testing*, because the facilities tested may interact with other facilities which are outside the scope of the benchmarking exercise.

2 Some Pitfalls in Benchmarking

2.1 Relevance

The requirement for benchmarking has permeated computer science, and is particularly important for vendors of software and hardware. Software benchmarking has reached its highest level of sophistication in the area of databases. Like programming languages, databases share some common functionalities, but new generations have new features which may negatively impact performance on some traditional functionalities (for example when moving from hierarchical to relational to object databases). An important reference for benchmarking is *The Benchmark Handbook* [Gra91] which, while it addresses the database area, has many lessons for benchmarking programming languages and constraint programming systems.

The first key criterion for benchmarking listed in the handbook is relevance. Another key criterion is simplicity, but this carries its own risks. As a case in point, the handbook considers the classic *mips* (millions of instructions per second) metric, concluding as follows:

> It is certainly a simple benchmark... The main criticism of the mips metric is irrelevance - it does not measure useful work.

We are reminded of an anecdote, which we hope is relevant here:

> In the middle of a dark night Fred encounters a drunken man who is frantically scouring the ground under a street lamp. “What are you looking for?” asks Fred. “I'm trying to find my keys!” - “And where did you lose them?” “Over there”, says the man and points into the darkness. “But why are you looking here then?” asks Fred, baffled. “Because I can see much better here in the light”, says the man.

While nobody has recommended *mips* or even *lips* (logical inferences per second) as a useful benchmark for constraint logic programming systems, the temptation to choose benchmarks that are easy to run, rather than relevant, has claimed some victims.

For CLP the classic benchmark has been the *N-Queens* problem. This problem is a standard example for finite domain constraints, and thus it is available as a demonstration program in almost every CLP system. The N-Queens problem is very uniform in that every pair of variables is constrained, and all the individual constraints are disequalities. As such it is a candidate for the CLP equivalent of naïve-reverse, used to measure *lips* for Prolog systems. However an even simpler problem requiring only search, disequality constraints, and backtracking, would be to find a list of n distinct elements.

Demonstration programs solving a number of other puzzles (mostly extracted from [Van89]) are circulated with many CLP systems, and it is therefore very easy to use them to benchmark the different systems. Several CLP developers have used sets of such puzzles to to validate new implementation ideas [COC97, DC93, MW96, Zho98]. In each case the benchmarks were used to validate some newly implemented design decisions in CLP systems.

A similar set of puzzles was also used by Fernandez and Hill in "A Comparative Study of Eight Programming Languages over the Boolean and Finite Domains" [FH00]. The aim of this paper was not, like the others, to present new implementation ideas for CLP, but to "aid others in choosing an appropriate constraint language for solving their specific constraint satisfaction problem".

Although in [COC97], the benchmarked puzzles are claimed to be "clean and fairly representative of real-world problems", there is no evidence for this. On the contrary, our experience of solving real-world problems at IC-Parc, some of which have been described in published papers [HEW$^+$98, RW98a, EW00, EW01] has shown that CLP puzzles are not at all representative of real-world problems. A similar conclusion can be drawn from reports of projects involving real-world applications at SICS and elsewhere [OAI$^+$98] and [BCD$^+$00]. Three typical features of programs solving real-world problems - incomplete search, linear constraint solving and global constraints - are completely absent from the benchmarks in [COC97, DC93, MW96].

The relevance of performance benchmarks on puzzles to the choice of CLP system for solving real problems was not discussed by Fernandez and Hill. Do the keys lie under this particular street lamp?

2.2 Comparing Equivalent Programs

Having decided to compare different CLP languages by time-trialling, it is necessary to write a program in each language to solve the same problem. The first requirement is that each program be correct.

The comparison by Fernandez and Hill [FH00] highlighted an awkward aspect of time trialling CLP systems. It is sometimes hard to write programs in two different CLP languages that have exactly the same specification.

This was exactly the error Fernandez and Hill made in programming the SRQ puzzle [Hen96]. The programs solving the SRQ puzzle in the different CLP systems are published on the web [Fer97]. We found many differences between the programs, and in particular programs we discovered:

- some constraints missing,
- some choice points incorrectly cut off

The results produced by the different programs were all correct, but the omissions itemized here could have a significant effect on program performance. (In the extreme case it would be easy to write a specific program for any single combinatorial problem that went straight to the answer without search.)

2.3 Comparing Equivalent Implementations

When time-trialling high-level programming languages, it is necessary to write high-level programs. As we pointed out above, it is possible when using most CLP languages to drop into a low-level language to "hand-code" particular algorithms or constraints. If the programs used for time-trialling exploit this facility then the benchmark is useless. The worst case is when the program in some of the languages being compared drop into low-level code, while the programs in others do not.

Fernandez and Hill also made this mistake in [FH00]. Some of the programs implementing the SRQ puzzle dropped into low-level **C** code, and others did not.

2.4 Publishing Source Code

Benchmarketing [Gra91] is the exploitation of benchmarks for other objectives. To minimise benchmarketing full disclosure of the programs used for benchmarking is essential. The reason we were able to identify the deficiencies in Fernandez and Hill's SRQ benchmark, was because the programs were published. Unfortunately none of the other programs used in Fernandez and Hill's comparison [FH00] were published. While it is possible to guess at differences in the programs used in different systems (for example the use of a global constraint in some systems but not in others), the failure to publish the programs makes it impossible to fairly assess the quality of the comparison.

We managed to retrieve the version of ECL^{i}PS^e (*3.5.2*, released from ECRC in 1994) that was used in the benchmarking of Fernandez and Hill. For the magic sequences problem [FH00] we wrote a simple program that, even for this version of ECL^{i}PS^e, was dramatically faster than the times reported in the benchmark tests. This, and all the ECL^{i}PS^e programs used for producing benchmark results reported in this paper, are available at
`www.icparc.ic.ac.uk/eclipse/CONSTRAINTS_code/`
Since the programs written in the different languages to solve this problem were not published, it is impossible to tell if the comparative timings on this benchmark were a result of the different system implementations, or merely a result of the different encodings, and consequently the different algorithms, used to solve the problem.

2.5 Extrapolating Benchmark Results to Different Hardware

Ideally, when one uses benchmark results to choose software to run on a given hardware configuration, the benchmarks would be run on that very hardware configuration. In practice, however, benchmarks run on one hardware configuration are used to make decisions about software to run on a different hardware configuration.

Sometimes benchmarks run on one configuration are used to predict the absolute performance of the same benchmarks running on another configuration. More often the comparative performance of two different systems on one hardware configuration is taken as a guide to their comparative performance on the target hardware.

Both kinds of extrapolation are risky. The most common pitfall in comparing language implementations is that a system may be "more" compiled on one machine than another (for example SICStus CLP is compiled to machine code on Sparc, but not on Intel machines). Clearly if system A is compiled to machine code on machine 1, and system B is compiled to machine code on machine 2, but not vice versa, their relative performance is quite different on the two machines. If the target hardware is machine 2, then running benchmarks for software systems A and B on machine 1 is a mistake.

A more subtle error is the attempt to compare two systems which do not run on the same hardware at all. Fernandez and Hill sought to compare the CLP systems SICStus and IF/Prolog with another set of CLP systems (clp(FD), CHR, ILOG, Oz and B-Prolog) without running the systems on the same machine [FH00]. Their approach was to run the ECLiPSe system on both machines, and then use the ECLiPSe performance on the different machines to normalise the results. Thus if system A ran half the speed of ECLiPSe on machine 1, and system B twice as fast as ECLiPSe on machine 2, Fernandez and Hill concluded that system A ran one quarter of the speed of system B.

This method of comparison is curious in that, by improving the performance of ECLiPSe on machine 1, the ECLiPSe developers could at a stroke improve the comparative performance of system B relative to system A without either system changing in any way! Kernighan and Van Wyk's [KV98] caveat that no benchmark result should ever be taken at face value, applies a fortiori in this case. To conclude anything about the comparative performance of two systems by running them on different hardware, it is essential to explore in detail what operations contributed to the performance of the different systems on the different machines. Recording numbers without analysis, as in [FH00], is either useless or misleading.

2.6 Parameter Settings

Even if two systems are benchmarked on the very same hardware, it is still possible to dramatically influence their performance by setting system parameters. One example [Gra91, p.303] is the level of performance monitoring: one system might have a much more complete, time consuming, tool than another. For CLP systems the level of debugging has a significant influence on performance. For time-trialling, all systems should run the same level of debugging, unless it is impossible to set the different systems to the same level. Assuming the purpose of the benchmark is to evaluate the speed of correct programs, rather than programs under development, debugging should be switched off.

Fernandez and Hill chose to run their benchmarks with the default parameter settings. ECL^{i}PS^e by default generates debuggable code while, for example, SICStus 3#5 which was benchmarked in [FH00] did not. The consequence was that the comparison published in [FH00] compared systems with different levels of debugging. The only justification for such an approach is that the benchmarking exercise is aimed at CLP users who are incapable of switching debugging on and off in the systems they are running. For such users, having debugging on as the default is arguably more important than performance, or any other aspect of the CLP language and its implementation!

Other parameter settings that influence performance and scalability include stack size limit, and garbage collection. Again Fernandez and Hill used the default setting, and again this had a significant effect on their benchmark results. For example they report that ECL^{i}PS^e *3.5.2* failed to solve magic sequences problems of size over 230. ECL^{i}PS^e, like many other CLP systems, has a number of parameters which ensure an application does not unintentionally consume all available swap space on a machine. By setting the ECL^{i}PS^e stack limit parameter higher, our program easily solved magic sequences problems of size 1000.

2.7 Timing Mechanisms

Kernighan and Van Wyk [KV98] reported that the timing services provided by programs and operating systems are woefully inadequate. Indeed correct timing is very hard to achieve with modern operating systems, especially when high-level languages are being executed. While elapsed time often includes an unpredictable overhead due to activities that are independent of the benchmarking, more precise measures of *CPU* time may exclude activities that *are* in the service of the program being benchmarked.

A typical example of an activity that is often excluded from timings is garbage collection. Time spent in garbage collection *was* included by the predicates used for timing in the version of ECL^{i}PS^e tested by Fernandez and Hill. On the other hand the SICStus timing built-ins never have included garbage collection. The published programs for the SRQ problem [Fer97] would, accordingly, time different activities. While Fernandez and Hill reported the surprising observation that "removing the garbage collection in other systems such as SICStus did not improve performance" they did not consider the possibility that their method might have been at fault.

3 Comparing CLP Systems - Feature Comparisons, and Benchmarks

3.1 Using Sets of Programs as Benchmarks

The use of a set of programs for benchmarking language implementations is not unusual. For Prolog a number of such program suites have been designed to test a broad range of language features. Examples are Pereira's AI Expert benchmark suite [Per87] and the Aquarius suite [Hay89]. Since programs are typically language-specific, different languages cannot be benchmarked in the same way - i.e. by time-trialling the same suite of programs in each language. Instead a standard set of problems can be benchmarked, by writing different programs to solve the problems in each language.

In fact benchmarking is a minor tool to use for comparing different languages. The comparison of programming languages for scientific computing by McClain [McC99], for example, concentrates largely on features. A report on benchmarking some standard computations is relegated to the end of the comparison. There are other similar comparisons in [Wac97].

Kernighan and Van Wyk's Timing Trials [KV98] of **C, Java, Perl, Visual Basic, Awk, Tcl, Limbo**, and **Scheme**, concluded that *if there is a single clear conclusion, it is that no benchmark result should ever be taken at face value*. They identified a few general principles:

- Compiled code usually runs faster than interpreted code: the more a program has been "compiled" before it is executed, the faster it will run.
- Memory-related issues and the effects of memory hierarchies are pervasive: how memory is managed, from hardware caches to garbage collection, can change runtimes dramatically. Yet users have no direct control over most aspects of memory management.
- The timing services provided by programs and operating systems are woefully inadequate. It is difficult to measure runtimes reliably and repeatably even for small, purely computational kernels, and it becomes significantly harder when a program does much I/O or graphics.
- Although each language shines in some situations, there are visible and sometimes surprising deficiencies even in what should be mainstream applications.

They encountered bugs, size limitations, maladroit features, and total mysteries for every language.

3.2 Comparing Algorithms Instead of Languages

In comparing CLP systems, we face several issues that have not arisen when comparing more traditional programming languages. The first issue is that the execution time for two programs in different CLP languages solving the same problem often depends less on the language implementation than on the precise algorithm executed by the program. We touched on this issue in the discussion of publishing source code above. However even when the source code is available, it is not always possible to determine whether two CLP programs in different languages will execute the same algorithm.

The benchmarked programs used to solve the N-Queens and other remaining puzzles in the different systems in [FH00] were not in fact made public by Fernandez and Hill. However they reported that the N-Queens programs in the study used depth-first search with a specific heuristic.

The shape of the search tree is strongly influenced by the heuristic for choosing the next variable to label. For the N-Queens problem the *first-fail* heuristic was used for many of the tests (reported in tables 1,2,3,4,5,6,7,9,11 in [FH00]). Unfortunately the authors did not report how the different programs dealt with tie-breaking, which as J. Schimpf pointed out to them [FH00] can change the shape of the tree, and consequently the execution time, by an almost arbitrary factor. As a result, the times reported in the tables were probably more influenced by their algorithm and its handling of tie-breaking, than by the implementation of the language itself.

Unit testing the disequality constraint would appear to be more informative than benchmarking the N-Queens.

3.3 Benchmarking Features not Shared by the Languages Under Comparison

Another issue in comparing CLP languages is that there is no standard set of *features* on which to compare them. Selecting a common subset of features - for example features supported by all finite domain constraint solvers - is a simple option. Unfortunately this kind of comparison would not only ignore, but actually count against an aspect of CLP that, we argue, is most important to its practical usefulness: the combining of solvers to support hybrid algorithms [RWH99].

An even worse mistake is to attempt to benchmark features that are not common to all the languages being measured. Fernandez and Hill set out to measure the cost of having omitted a feature from one language by benchmarking it against other languages with the feature. Their method was to choose a problem whose solution required, to their belief, the use of a particular feature.

The feature investigated by Fernandez and Hill was *reified constraints*. The problem they chose to benchmark reified constraint was self-referential quizzes (SRQs) [Hen96]. Two formulations of the problem were presented, and for each CLP language the code for each formulation was presented. Fernandez and Hill argued that better support for reified constraints rendered the encoding simpler and more natural. They also compared the performance of the all different programs, for each language and each formulation.

They admitted that another formulation of the problem [Fer97] which did not require reified constraints had been pointed out to them earlier [FH00, Note 1, p 281]. Although the programs executing this formulation proved to be more efficient than programs running the other two formulations, they did not use it in the comparison.

A benchmark can only serve to measure the importance of a programming language feature if applications can be found for which this feature is needed, and consequently languages not supporting this feature incur a performance penalty. The existence of a simple formulation which allows the problem to be solved more efficiently without this feature completely undermines the validity of the benchmark.

From a methodological point of view the right way to benchmark a particular functionality is by unit testing.

3.4 Benchmarking Algorithms

To benchmark how efficiently different languages support the same features, it is necessary to write programs in the different languages that run the same algorithm. The operations research and artificial intelligence communities have a strong tradition of analysing and benchmarking algorithms, and in particular the CSP community analyses the kinds of algorithms run by CLP programs.

For our benchmarking the analysis of algorithms must be sufficiently concrete to ensure that our benchmarks compare like with like. Three measurements often used in analysing CSP algorithms are [Nad89]:

- backtracks
- constraint checks
- space requirements

While these measurements are uncontroversial when measuring two different algorithms implemented in the same programming language, they can be very misleading when used to compare algorithms written in different programming languages.

3.4.1 Measuring Backtracks

When measuring backtracks two important details must be made explicit:

- Are *shallow* backtracks counted? These are backtracks that result from a propagation failure after the system has attempted to instantiate a variable to a value. Some CLP systems (such as ECL^{i}PS^e) will not include such an attempted instantiation as a search step.
- Are backtracks correlated with choice points? If a search tree has depth n, then after returning the last leaf of the tree a CLP system will typically have no more remaining choice points. However there remain, arguably, n more backtrack steps from the leaf to the root node, before the search tree has been fully explored. It is quite common when analysing algorithms expressed in CLP to count choice points instead of backtracks. Choice

points may reflect more accurately than backtracks the computational requirements of the algorithm. We note that the addition of CLP code to monitor the number of choice points in a CLP program can easily introduce new choice points into the code, and thereby changing the thing being measured.

3.4.2 Measuring Constraint Checks

It is even more tricky to measure constraint checks. When analysing CSP algorithms constraint checks are often assumed to take a fixed time per check [MF85]. However the efficiency of constraint checking is an important aspect of CLP implementation. Moreover most CSP analyses assume constraint definitions are held in tables. However most CLP benchmarks (and applications) use constraints whose definitions are implicit: often they are numerical or are defined as logical combinations of other constraints.

For the purposes of comparing languages two algorithms are the same if they make the same number of checks on each kind of constraint. For benchmarking it is best if a problem only involves one kind of constraint, in which case the relative performance of two languages will reflect their search performance on constraint checking for that kind of constraint. If there are many kinds of constraint in a problem, the performance may depend upon the ratio of different kinds of constraint checks. For example one language may be very efficient at checking numerical ordering constraints, and another more efficient at disequalities.

A more serious difficulty is that few CLP implementation support a facility to count constraint checks. Typically it is only possible to count a behaviour, such as *waking* events which more or less approximate constraint checks. Unfortunately the approximation may deviate wildly for certain programs, which happen to execute propagation sequence without raising events. If code is added to a CLP system to count explicitly constraint checks, this may seriously impair its performance.

3.4.3 Measuring Space Requirements

The space requirements of an algorithm may depend upon:

- domain information
- information about support for a domain value
- information supporting intelligent backtracking
- nogoods

To compare like with like, we must ensure that the algorithms are recording the same information. In particular, languages that support multiple solvers may record information which supports the interaction between solvers, and which may not be necessary for single-solver problems.

Certain choices, such as whether to record the upper and lower bounds of a domain explicitly, or only to record the set of values in the domain, are arguably properties of a CLP language implementation and not of any algorithm written in that language.

It is difficult to tease out what level of abstraction in the measurement of time and space requirements would be appropriate in order to compare CLP languages. For example, what should be deduced from comparing a system which implements AC3 with one that implements AC4? There is a risk of confusing a comparison of two language implementations with a comparison of two algorithms, but if neither implementation offers the other algorithm as an alternative, then we cannot exactly compare like with like.

Ultimately we must keep in mind Kernighan and Van Wyk's conclusion, that no benchmark result should ever be taken at face value. To make it possible for users of the benchmark to interpret the results it is necessary to record all the assumptions and "wrinkles" in the benchmarking process.

3.5 Summary

For software systems such as databases whose functionalities are well circumscribed it is possible to specify benchmarks in the form of *workloads* that reflect certain usages of the system. As database technology advances, from relational to object-oriented for example, the benchmarks are updated, perhaps a decade after the first research prototypes of the new technology have been introduced [Gra91].

CLP is changing quickly. Finite domain constraints have indeed been supported by CLP systems for a decade, but their implementation depends crucially on their interaction with other solvers, in particular linear constraint solvers. Benchmarking a finite domain system in isolation is useful only to potential CLP users who have no need for any other solvers than finite domains. It is unclear which potential users might have such restricted needs.

The interaction of finite domain and linear solvers is currently a topic of research so it is still early to establish benchmarks. To date any benchmarking in this area (e.g. [RW98a, Ref99]) has been aimed at comparing algorithms and not implementations.

It can be useful to test certain common functionalities of languages or systems in order to find out what price is being paid by each system for its additional functions, generality or power. In this case the comparison may record the benchmark results on the common functionalities and include some analysis of the interaction between the benchmarked features and the other parts of the language or system.

However it would be foolish to use a benchmark on finite domain constraints to "aid others in choosing an appropriate constraint language for solving their specific constraint satisfaction problem" [FH00].

4 CHIC-2 Benchmarking

In this section we report on a benchmarking exercise which started within the Esprit project CHIC-2 [CHI99]. The project investigated the application of CLP to industrial Large Scale Combinatorial Optimisation (LSCO) problems.[1]

4.1 Purposes of Benchmarking for CHIC-2

In CHIC-2 it was established that LSCO problems are *hybrid*, involving different kinds of constraints (continuous and discrete, temporal and spatial, constraints on tasks and on resources, etc.). The need for hybrid algorithms for solving LSCO problems was tested and proven in the course of the project. We concluded that it was more important to ensure the programming platform has the right amount of flexibility and facilities to allow the programmer to program the right approach easily, rather than ensuring that a particular feature is optimised for performance, especially at the expense of flexibility.

The aim of the benchmarking was to quantify the facilities of ECL^{i}PS^e for addressing LSCO problems. The objective was to provide some measurements that would truly reflect the value and usefulness of the different ECL^{i}PS^e facilities for modelling and solving such problems.

This objective immediately raised the classic difficulties of benchmarking: what could be quantified, and how would the measurements shed light on the performance of the software being benchmarked?
The CHIC-2 benchmarking sought to measure:

1. the performance of LSCO applications running under ECL^{i}PS^e,
2. the performance of various features of ECL^{i}PS^e, and
3. the suitability of ECL^{i}PS^e as a platform for hybrid algorithms.

For these purposes, we designed and ran several benchmarks. Two of them, in particular, not only revealed aspects of the ECL^{i}PS^e platform, but also hold some lessons about the benchmarking process itself. We therefore discuss just these two benchmarks in the current paper.

Firstly we performed *unit testing* on some standard constraints, in order to assess the cost and benefits of making ECL^{i}PS^e flexible and rich enough to support hybrid algorithms for LSCO. To assess the costs, we compared the performance of some simple finite domain constraints in ECL^{i}PS^e with those supported by other systems. To explore the benefits we made a comparison of two alternative solvers available in ECL^{i}PS^e. This kind of benchmark provides

[1]The European ESPRIT project CHIC-2 had four elements: the solution of four demanding, real-world planning/scheduling problems, the development of hybrid algorithms to better support the solution of large scale planning and scheduling problems, the development of a methodology to help reduce the cost of developing such applications, and the enhancement of the ECL^{i}PS^e platform to fully support the efficient implementation and execution of the results of the project.

some of the background knowledge the ECL^{i}PS^e user might need to decide which solvers to use for handling a particular problem most efficiently.

Secondly we ran an *application benchmark*, designed to measure development time and effort, and the maintainability of the resulting program. We chose a staff rostering problem, introduced by another CHIC-2 partner, and we compared ECL^{i}PS^e with two other constraint programming platforms.

The two benchmarks are discussed in the following sections.

4.2 Simple Constraint Propagations

The unit tests benchmarked in this section were used to compare

- The same solver (fd) running under two different CLP systems
- Two different solvers, running under the same system (ECL^{i}PS^e)

In the first exercise the two systems whose fd solvers we compared were ECL^{i}PS^e (version 5.2) and SICStus (version 3.8.5). The ECL^{i}PS^e tests were compiled with reduced debugging information, and for SICStus the tests were compiled into *compactcode*. The SICStus timer *excludes* time spent on such activities as garbage collection and stack expansion. By contrast ECL^{i}PS^e's timer *includes* these activities. To partly compensate, garbage collection times were added back to the SICStus timings.

In the second exercise, the two solvers, running under ECL^{i}PS^e 5.2, were *fd*, and an external linear solver accessed via an ECL^{i}PS^e interface. The external solver was CPLEX, version 6.5.

All tests were run on the same hardware and operating system - specifically a 933MHz Pentium III with 512 Mb of memory, running Linux.

4.2.1 The Unit Tests

We designed a set of unit tests, to evaluate the scalability and performance of the finite domain and linear solvers on a class of very simple constraints.

In the unit tests, the constraints had the form $X \geq Y + B$ where X and Y were integer variables, and B was instantiated either to 0 or to 1. The effect of setting $B = 0$ or $B = 1$ is discussed in the section 4.2.3 below. Each test generated $M - 1$ constraints in M variables, each variable having domain $0..M$, and each constraint having the form $X \geq Y + B$.

To aid our discussion of these constraints in the following sections, we will talk about $X2 \geq X1+B$ as the "first" constraint, $X3 \geq X2+B$ as the "second" and so on.

For the benchmark we ran a sequence of tests of increasing size M: from 200 to 10,000. The constraints imposed an ordering on the variables, e.g. if $M = 5$, $B = 1$ the constraints were:

$$0 \leq X1, X2, X3, X4, X5 \leq 5,$$
$$X2 \geq X1 + 1,\ X3 \geq X2 + 1,\ X4 \geq X3 + 1,\ X5 \geq X4 + 1.$$

4.2.2 Comparisons of ECL^{i}PS^e's and SICStus' fd solvers

In the first unit test a program was written to generate, and post, the constraints in order (written *up* in the graphs in this section). Thus constraint $X2 \geq X1+1$ was added first, then $X3 \geq X2+1$, and then $X4 \geq X3+1$ and so on. The program had to be slightly altered to run in SICStus, because of minor syntactic differences between SICStus and ECL^{i}PS^e. It was not expected that these differences would lead to big performance differences.

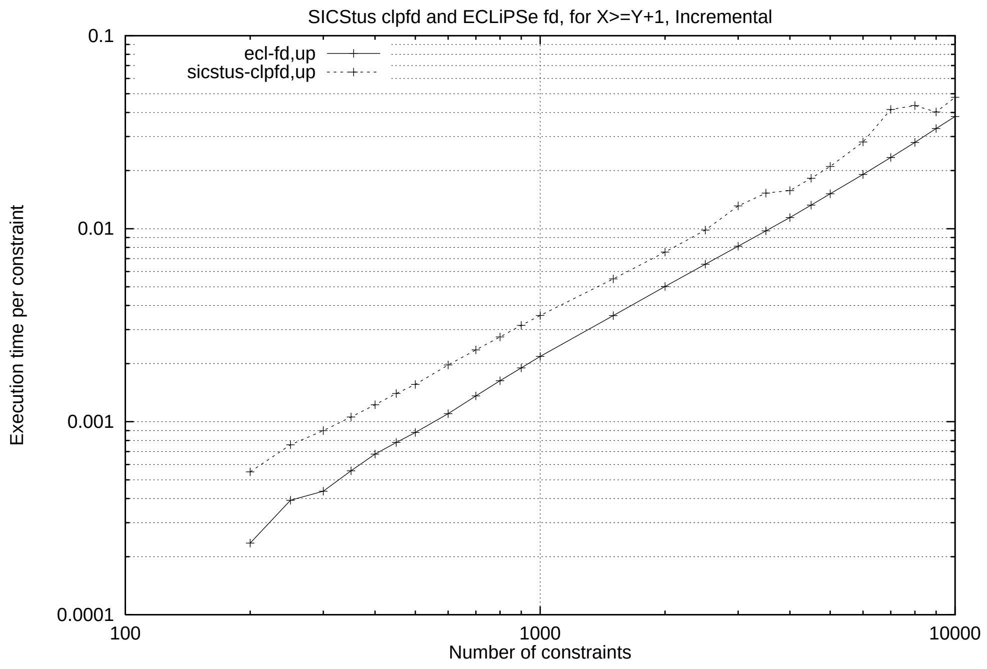

Figure 1: Comparing fd in ECL^{i}PS^e and SICStus

The graphs in this section use a logarithmic scale on both axes. A horizontal line would indicate that the execution time per constraint was constant. Steeper lines reflect, linear, quadratic and increasingly higher order polynomial relationships between the number of constraints and the time per constraint. In the graph in Figure 1 the time per constraint increases approximately linearly with the number of constraints both for SICStus fd and for ECL^{i}PS^e fd. If the lines were exactly parallel, this would represent a constant factor difference between the two solvers.

Accordingly, the graph appears to show that the performance of the ECL^{i}PS^e fd is around 50% faster than the SICStus fd solver. However the lines converge slightly, indicating that the difference narrows as the number of constraints grows.

Adding Constraints in a Different Order Adding constraints in increasing order is actually the worst case, because each time a constraint is added, new upper bounds are propagated back to every previous variable.

We therefore explored the effect of adding constraints in random order, and in various specially constructed orders. One interesting case was the *odd/even* order: odd-numbered constraints were added first, and then the even-numbered ones, thus: $X2 \geq X1+1, X4 \geq X3+1, X6 \geq X5+1 \ldots, X3 \geq X2+1, X5 \geq X3+1, \ldots$

Another case was the *binary-chop* ordering. This order maximised the 'distance' between each constraint posted. For a chain of 10 variables, $X1 \ldots X10$, the constraint $X6 \geq X5+1$ was posted first, followed by $X3 \geq X2+1$ and then $X8 \geq X7+1$ and so on.

The results are shown in Figure 2.

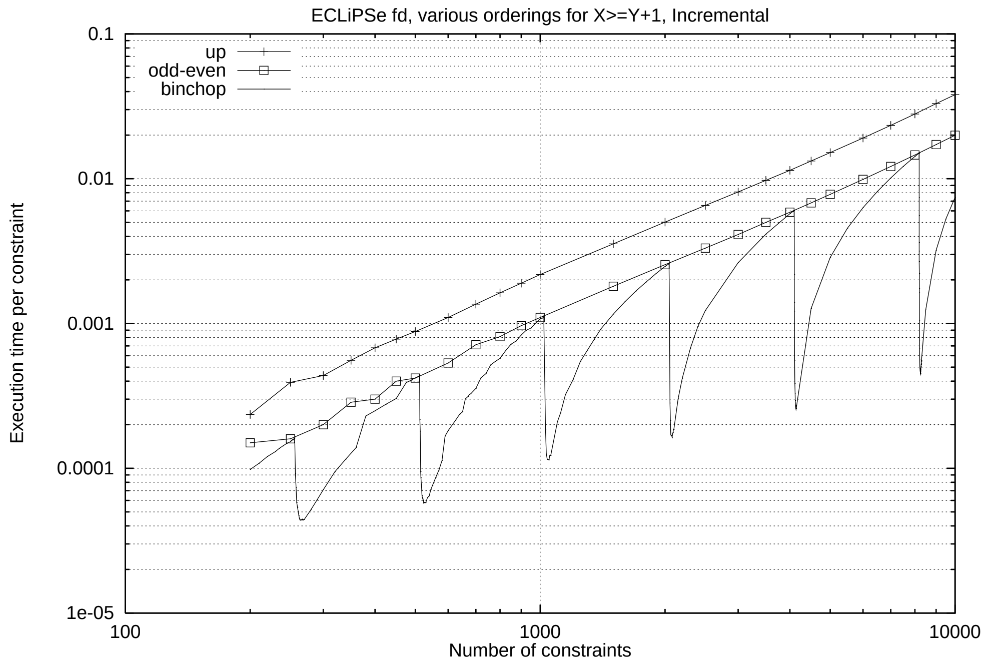

Figure 2: Result of changing the order of propagation - ECL^{i}PS^e

The execution times for both new orderings are much faster than for the original ordering. The *odd/even* ordering is consistently about twice as fast. The binary chop ordering is also faster - sometimes much faster. Its highly erratic behaviour is very evident: the performance improves dramatically when the number of variables is any power of two.

Most important of all is that the difference in performance due to the order of posting constraints is far greater than the difference between the two systems SICStus and ECL^{i}PS^e. Indeed the difference in performance between the test with $2^N - 1$ variables and 2^N variables in the binary-chop ordering far exceeds the average differences between the orderings themselves.

To check if this erratic behaviour was specific to the ECL^{i}PS^e fd solver, we tried the same orderings on SICStus fd solver. The results are shown in Figure 3. The behaviours of the two systems are indeed very similar.

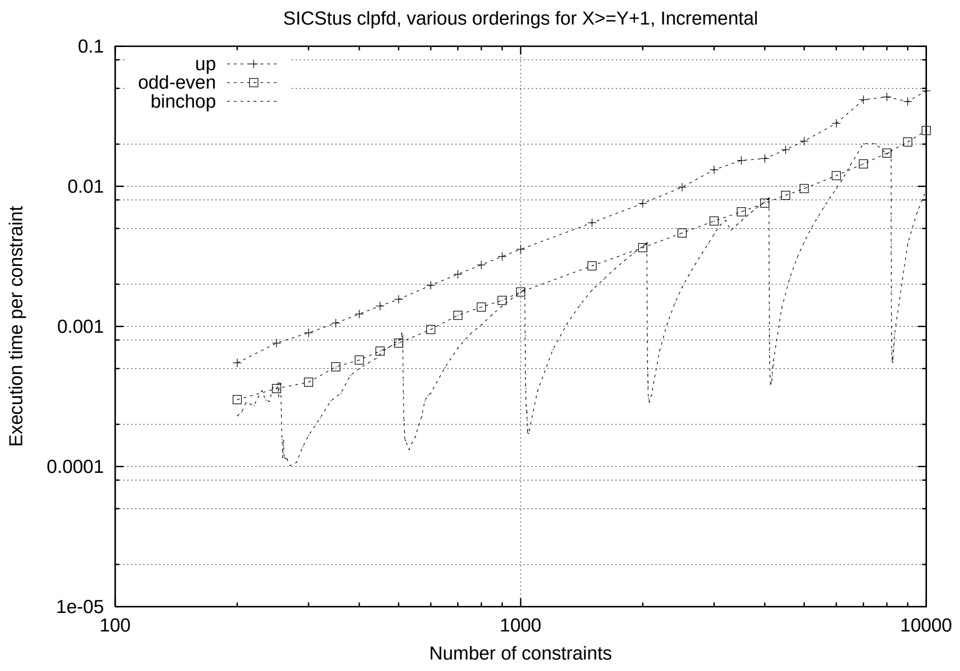

Figure 3: Result of changing the order of propagation - SICStus

Postponing Constraint Propagation ECL^{i}PS^e offers the programmer control over the priority of different activities. By raising the priority of the routine which posts constraints, for example, the programmer can ensure that all the constraints are posted before any propagation takes place.[2]

The consequence of changing the priority in this way is that the total number of propagation steps executed by the system may be dramatically reduced, though the domains of the variables are reduced by the same amount. To explore this effect we ran the same unit test as before, posting the constraints in increasing order, and examined the effect of changing the priorities in the way just described. Thus we compared the *incremental* case, where constraint propagation was performed after adding each constraint, with the *batch* case,

[2]The ability to control the propagation behaviour has proven to be useful in real applications[WS02, EW00], but is not available in other CLP platforms that do not have execution priorities.

where propagation was postponed until all the constraints had been added. The results are shown in Figure 4.

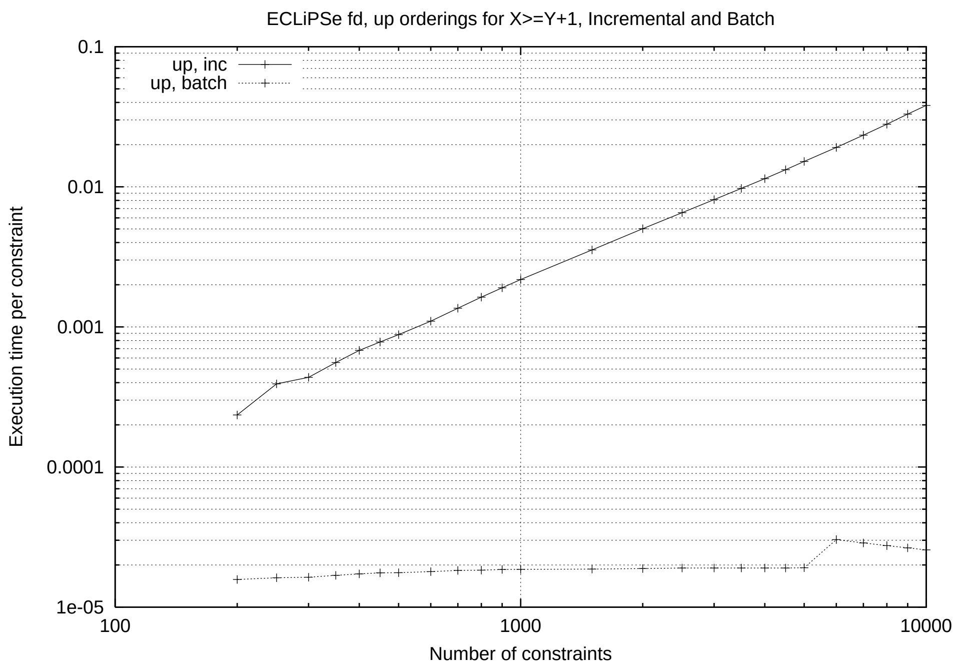

Figure 4: Comparing incremental and batch propagation behaviour

The effect on performance is again dramatic. The time per constraint for the batch case is almost constant.[3]. The difference between incremental and batch propagation again exceeds that of any of the variations in the unit test explored so far. However, the order of posting constraints is still a very important factor even in the batch case. The next graph in Figure 5, shows the execution times of the batch case for the odd/even and binary chop constraint orderings. In fact the odd/even ordering has almost identical performance for both the incremental and the batch cases. The binary chop has erratic performance, as before, but now when the number of variables is a power of two, it is not the best case but the worst case!

Clearly different scheduling of the propagation steps within a propagation sequence can yield very different results, because the order of propagation steps can have an important effect on the *number* of propagation steps needed to complete the sequence. For example, by changing the order of propagation steps, the number of steps in the propagation sequence may increase from a linear multiple of the number of constraints to a quadratic function of it.

[3]The kink at around 6000 constraints in this and subsequence graphs is due to garbage collection

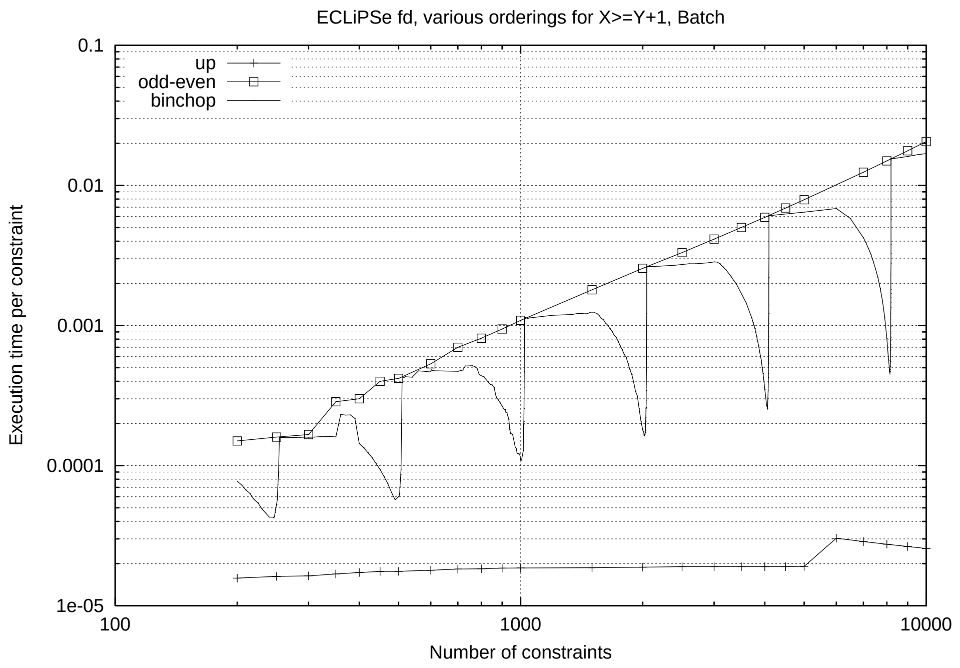

Figure 5: Result of changing the order of propagation in batch mode

Finally, when adding the constraints in a random order, we observed that the propagation sequences were typically well-behaved, tending to increase linearly in proportion with the number of constraints. This was also reflected in the computation times.

These results demonstrate how sensitive execution times can be to the order of constraint posting, and the exact scheduling order of the propagation steps. ECL^{i}PS^e schedules the most recently woken goal first. Changing to another policy could dramatically improve performance on this worst case, but make other cases much worse. For a simple case like this unit test, we could establish an optimal scheduling policy, but this policy might not be the best for other benchmarks, or for real applications.

This benchmark highlights the difficulty of comparing CLP systems: small differences in the implementation of the scheduling could lead to big differences in performance.

4.2.3 Comparison of ECL^{i}PS^e fd and eplex

Strict Inequations The same unit tests discussed in the previous section were used for comparing two different solvers available via ECL^{i}PS^e. These solvers were fd and an external linear solver, supported via the ECL^{i}PS^e *eplex* library. The eplex library offers interfaces to two alternative commercial linear and mixed integer programming packages, XPRESS from Dash [Das01] and CPLEX from ILOG [ILO01]. The solver used in the current comparison was CPLEX. In using the CPLEX solver, an objective function must be specified, and for the reported benchmarks this objective was minimisation of the first variable.[4]

An ECL^{i}PS^e programmer is often faced with a variety of algorithmic choices for solving an application. A typical example is the choice between propagating lower and upper bounds on a set of variables using the fd solver, or finding lower bounds on a cost function using a linear solver. Another choice might be to apply both solvers, or apply different solvers to different subsets of the problem constraints and variables.

Unfortunately it is not currently possible for the software to automatically apply the best combination of solvers, nor do we believe this will be a realistic possibility for the foreseeable future. Moreover the number of ways of extracting subproblems and combining solvers is very large so it is not possible to blindly try out every alternative and choose the best.

The programmer therefore needs some understanding of the relative performance of different solvers, to judge whether, for example, the computing time required to extract a cost lower bound from the linear solver for a particular subproblem might be worth the benefit in the context of the whole problem solution. The objective of the following benchmarking exercise is to help with this kind of understanding.[5]

While finite domain solvers and linear solvers handle different classes of constraints, the constraints used in the benchmark are, of course, handled by both kinds of solver. Moreover this class of constraints was chosen to minimise the differences due to the different kinds of consistency enforced by the different solvers. Specifically, the linear solver enforces global consistency for the linear relaxation of a set of constraints. For this class of constraints fd also establishes global consistency. On the other hand an fd solver enforces integrality on its integer variables. For this class of constraints, the optimum returned from the linear solver is also integer valued.[6]

In section 4.2.2 above we distinguished two kinds of constraint solving: *incremental* and *batch*. Traditionally fd solving has been performed incrementally, propagating the results of each new constraint before adding the next one. On the other hand linear constraint solving has been executed in batch mode, adding all the constraints and then running the solver once on the complete set.

While XPRESS and CPLEX now both offer flexible support for incremental addition of constraints, there is an extra overhead in the ECL^{i}PS^e eplex interface of passing the results from the external solver back to the ECL^{i}PS^e engine. The benchmarks presented in this section were designed to explore the costs of

[4]Maximisation of the first variable was also tried, and the results were similar.

[5]It is not aimed to measure the performance of the particular external solver, as a stand-alone package, nor is it intended to objectively compare finite domain constraint solving with linear constraint solving. Consequently there is no attempt to measure the overhead of either solver due to the ECL^{i}PS^e interface.

[6]This is because the constraints added in the benchmark are *unimodular* [Wil93].

invoking a linear solver both in batch mode and incrementally.

Incremental processing in eplex was accomplished by passing the constraints one at a time from ECL^{i}PS^e to the external solver, and running the solver each time. Each time the external (CPLEX) solver was run it was *warm started* using the previously solved matrix, with an added row for the new constraint, and starting from the previous solution's *basis*. For batch processing, the external solver was only run once, after all the constraints had been added. The results are shown in Figure 6.

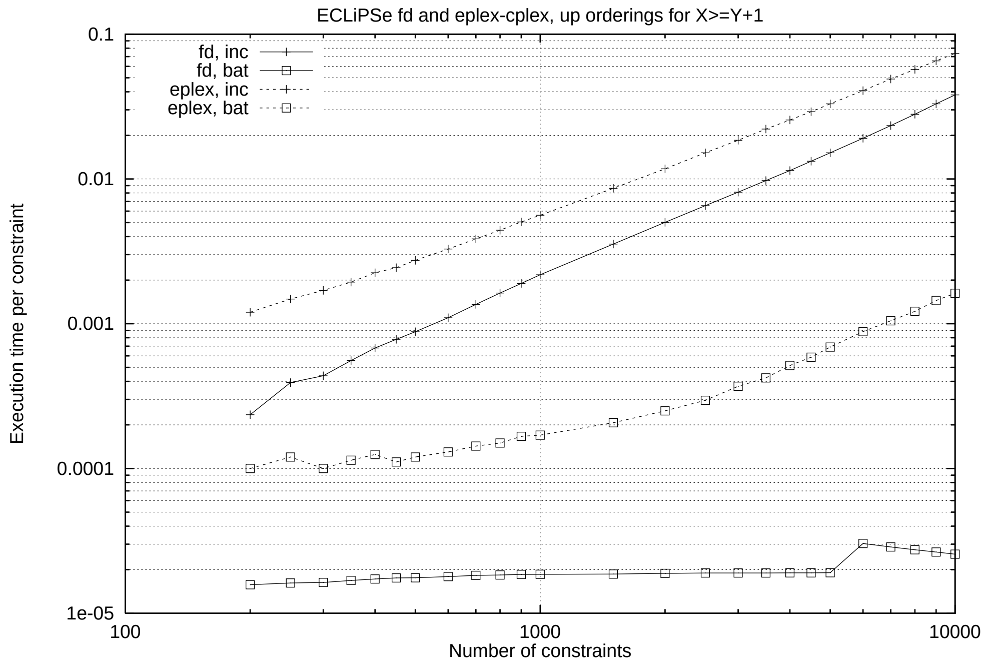

Figure 6: Comparing Strict Inequations in ECL^{i}PS^e fd and eplex

As expected, the batch simplex performed much better than the incremental simplex, because the external solver was called once in the batch case, and as many times as there were constraints in the incremental case. In fact the batch mode for both solvers outperformed the incremental mode for both solvers.

However, the obvious conclusion - that the difference between batch and incremental mode is more significant than the difference between the solvers - can be quickly contradicted by adding the constraints in a different order, as we saw in section 4.2.2 above. For completeness, we also tried the binary chop ordering with CPLEX. However, although the changed order resulted in visible differences in the execution times, the differences were less than 10%.

Non-strict Inequations The performance of an fd solver is highly dependent on the amount of propagation which results from adding each new constraint.

In the unit tests discussed so far, the inequations ($X \geq Y + B$ with $B = 1$), are strict, and if X and Y start with the same upper and lower bounds, then one bound of each variable must be tightened as a result of propagating this constraint.

Setting $X \geq Y + B$ with $B = 0$ generates non-strict inequations of the form $X \geq Y$. If X and Y start with the same bounds, then no tightening is achieved as a result of propagating this constraint.

The difference in the constraints makes little difference to an external linear solver which uses a variant of the Simplex algorithm. This is reflected in the experimental results.

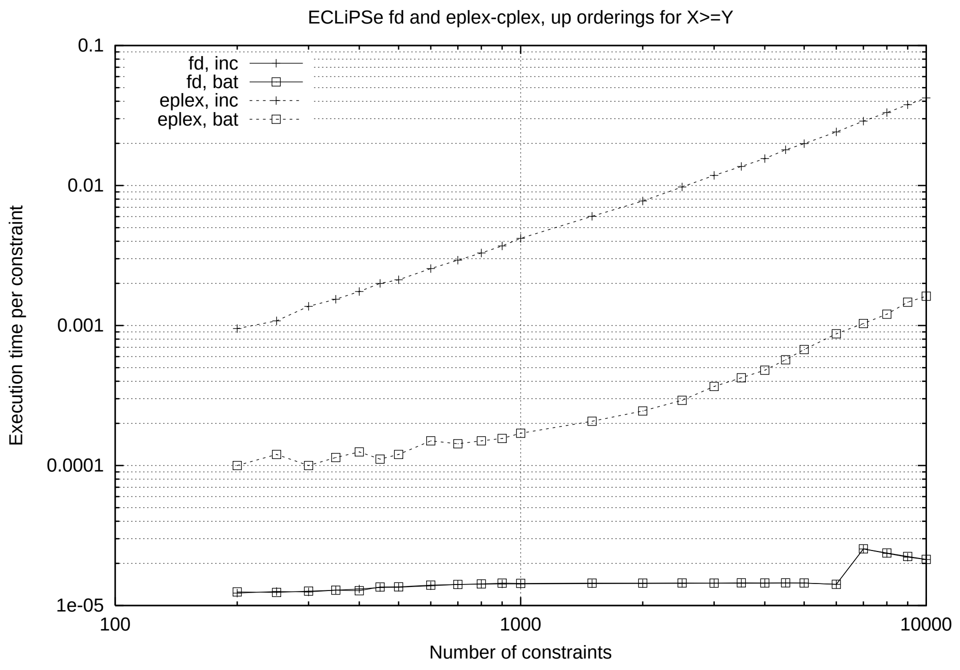

Figure 7: Comparing Non-strict Inequations in ECL^{i}PS^e fd and eplex

For a finite domain solver, however, $X \geq Y$ and $X \geq Y + 1$ require very different handling. Adding a new constraint of the form $X_{n+1} \geq X_n$ has no effect on the domains of any of the previously defined variables $X_i : i = 1..n$, and no propagation takes place. In contrast, the $X_{n+1} \geq X_n + 1$ case is quite costly, because each new constraint causes a propagation sequence which reduces the domain of every previous variable $X_i : i = 1..n$.

The tradeoff between fd and eplex is therefore influenced most of all by the amount of propagation that has to be performed at each step. In case a problem is "loosely" constrained, the fd solver is likely to dramatically outperform a linear solver.

Other Variations on these Unit Tests Even though the class of constraints we have studied in the above unit tests are extremely simple, there are many variations that can influence benchmark results. Another factor that has not yet been considered is *failure*: what happens if the set of constraints is inconsistent.

Clearly this is the main reason to support incremental constraint solving. If new constraints are generated in the context of a larger algorithm, especially if the program performs a great deal of computation before adding each new constraint, then detecting an inconsistency before all the constraints have been added can bring an arbitrarily large performance benefit.

In the case that the full set of constraints are inconsistent, the relative performance of the batch fd solver and the eplex solver running in batch mode may be quite different from the results shown above in Figure 6.

We performed a number of tests to explore the effects of inconsistency. For the incremental ordering, the fd solver proved faster than eplex independently of the stage m at which an inconsistency, of the form $X_1 \geq X_m + 1$, was introduced. Relatively, however, the gap between the fd solver and eplex was reduced. The behaviour of fd, however, would be highly dependent on the original size of the variable domains.

There are many further variations of the above unit tests. In particular in the tests described above, the constraint graph has the form of a single chain. By changing the variables involved in the different inequations, a huge variety of different constraint graphs could be produced, yielding binary trees, consistent and inconsistent cycles, and so on. Moreover each new variation can be combined with variations discussed above. Thus, even for a very simple class of constraints, a "representative" set of unit tests is extremely difficult to construct.

Some Remarks for ECL^{i}PS^e Users There is some overhead associated with sending the constraints to an external solver in the case where the constraints are simple to handle. The benchmarks also show that incremental constraint solving can be computationally expensive for both kinds of solver. The need to detect inconsistencies incrementally has been well established for finite domain constraint solvers. Incremental behaviour of the linear solver is also important, particularly when both the fd and linear solvers are used together, communicating information at every search step [RW98b, EW00].

4.2.4 Conclusions

Even unit testing does not provide a very satisfactory basis of comparison between different constraint solvers and systems. While linear and finite domain solvers do share a common subclass of constraints for which they both support complete decision procedures, comparing their performance on the common subclass, as described in section 4.2.3, is unsatisfactory.

The reason is that neither solver is designed for that particular subclass: the algorithms used in each solver are designed to handle much larger classes of constraints. Benchmarking the solvers on a specific subclass does not do justice to either of them.

Even where the solvers are more similar, as in the comparison of section 4.2.2 above, the results may be highly influenced by a minor algorithmic detail, such as a particular order in which propagation steps are performed in a propagation sequence. The same order might be ideal in one propagation sequence, and have worst case behaviour in another.

However even between the finite domain solvers in ECL^{i}PS^e and SICStus, the algorithms will be different as they are designed for different purposes. For example the handling of priorities in ECL^{i}PS^e incurs extra overhead. This overhead has benefits which might not be revealed by unit testing on the common subset of the different solvers.

There is a third difficulty in using unit tests to compare solvers, and that is to decide how many unit tests provide a "reasonable cover" of the functionality of the solvers. If the number of unit tests is very large, this presents a barrier in terms of the time and effort required for benchmarking, and it also makes the results of the comparison impenetrable. Thus it happens very often that a few unit tests, such as those described in sections 4.2.2 and 4.2.3 above, are assumed to reflect the functionality of the whole solver. Clearly this is far too small a number to give any realistic basis for comparing solvers.

The tests suggested that the overhead of supporting priorities in ECL^{i}PS^e were not prohibitively costly in terms of the run-time performance of the system, at least in the case where all the constraints had the same priority. These results were of some use for the system developers, but clearly they could not have been employed by potential users of the systems to decide between them.

4.3 Case Study: Rostering Problem

The next benchmark sought specifically to provide some measure of programmer time: both for program development and program maintenance.

There has been a previous comparison of CLP languages by setting different programmers the task of solving the same problem in different languages [BBV$^+$97]. The difficulties they encountered in achieving a convincing comparison between the languages were similar to those reported in this section.

The activity benchmarked was the development of a program to solve a given problem. The intention was to compare:

- the time needed to develop the program,
- the size of the resulting program and
- the run-time performance of the resulting program.

The size of the program was intended to give some objective measure of how difficult the program was to write, maintain and enhance.

One problem was used for this benchmark, and a solution for the problem was developed on three different constraint programming platforms. The benchmark exercise was carried out by just one person on each platform. The platforms compared in this benchmark were ones used by different partners in the CHIC-2 project.

When carrying out this benchmark the CHIC-2 project partners were acutely aware of the limitations of the exercise. A thorough benchmarking exercise would have required:

- a large number of "comparable" program developers and
- a wide variety of problems.

It would also have been of interest to have tested a wider variety of platforms.

Because the sample was so small (i.e. one problem and one developer), the partners recognised that the results could not be used for drawing any serious conclusions about the merits of the different platforms for developing solutions to LSCO problems.

The partners recognised that *this situation is not uncommon.* In practice, the resources needed to carry out manpower intensive benchmarking exercises of this kind are rarely available. The alternative is not to make any comparisons at all.

This benchmark is accordingly presented with full acknowledgement of its limitations. The results should be taken more as a case history than as a scientific benchmark.

4.3.1 Problem Definition

The objective of the problem is to cover a weekly work load with morning, day and evening shifts. The shifts must be arranged in a grid which meets a variety of constraints. For each day of a typical week, the number of shifts per type is given. Some supplementary shifts are added in order to manage absenteeism. Last but not least, the days off must be planned. For each shift, the possible labels are:

M	Morning shift
J	Day shift
E	Evening shift
K	Supplementary shift
R	Rest Day

A solution, for N workers, is an N line grid of weeks specifying shifts and days off for one employee. The rosters for the other employees are obtained by starting at the second, third, etc. lines of the grid, which describes a continuous rota.

The constraints and objective are summarised as follows:

- **Hard constraints.**
  The first constraint is the number of labels for every day of the week. For each day, all N labels must be set. Secondly there are constraints on the labels which must be assigned for each day of the week. Thirdly there is a constraint on the length of a working period: no more than 6 consecutive days without a day off. The fourth constraint is not to have more than 3 days off consecutively.

- **Flexible constraints.**
  The soft constraints are not to have isolated days off and not to have morning shifts after evening shifts.

- **Objective.**
  The program's purpose is to minimise the cost, which is defined as the number of violations of soft constraints.

A sample solution with cost 1 for a 5-week problem is:

Grid	Mon	Tue	Wed	Thur	Fri	Sat	Sun
Week1	R	R	M	E	R	R	R
Week2	M	R	R	J	M	M	R
Week3	R	E	J	J	E	E	R
Week4	R	E	J	K	J	M	R
Week5	E	J	K	K	J	J	R

Table 1: A Five Week Roster

4.3.2 The Platforms

Solutions were developed on three platforms:

- $\mathrm{ECL}^i\mathrm{PS}^e$ 4.1 with its finite domain library,
- Claire 2.3 with the Eclair finite domain library (Bouygues in-house) and
- ILOG OPL with ILOG Solver (commercial).

All three platforms make similar claims and are thus comparable. They:

- support elegant and declarative modelling,
- interface to similar solvers and
- allow programming of search heuristics.

The purpose of this comparison was therefore not to prove that high-level tools are useful (this was implicitly accepted), but to compare $\mathrm{ECL}^i\mathrm{PS}^e$ with a commercially available tool (OPL) and a platform that has invested more development effort into efficient compilation (Claire).

The (different) ECL^{i}PS^e solutions were developed by (different) experienced ECL^{i}PS^e users with access to the ECL^{i}PS^e developers. The Claire solutions were written by a senior developer with some experience of Claire and access to Claire developers. The OPL code was developed by a less experienced user not involved with the language development group.

4.3.3 Comparison

Development time One of the initial ideas was to compare development times. Due to lack of controlled conditions, the difference in experience with the platforms, the different programming experience of the developers and the small sample size, this comparison can only be very informal. Both the ECL^{i}PS^e and Claire developers got results within about one day. This includes modelling from the problem specification and the development of an initial search strategy that solved one problem instance. More time was spent later on improved search strategies and the solution of more difficult problem instances.

Because the developer of the OPL solution was less experienced, and did not have access to the language development group, it was deemed inappropriate to report on the time taken to develop the OPL solution.

Platform facilities All three platforms encouraged experimentation, and a significant number of strategies and variants were developed by all participants. The best solutions on the different platforms (within the development times reported above[7]) are called in the following:

- Sol-O (the best OPL solution)
- Sol-C (the best Claire solution)
- Sol-E (the best ECL^{i}PS^e solution)

Sol-O and Sol-C were reproduced in ECL^{i}PS^e. This reproduction caused no problems, as all the necessary facilities were supported by ECL^{i}PS^e. Sol-E was then reproduced on the other two platforms.

Code size The table 2 compares source code sizes on the different platforms.[8] The benchmarkers tried to follow the programming style of the original program, although there are differences in the modelling. The numbers given in the table are program tokens. A token is defined as a lexical unit of the corresponding programming language (e.g. an identifier, a number, an operator, a parenthesis, etc.). Comments in the code are ignored. This method of measuring has the advantage of being less sensitive to different coding styles. In particular, the length of identifiers, the amount of comments or the way lines are broken up has no impact on the result.

[7]Naturally the developers went on to build more sophisticated solutions. No attempt was made to translate these solutions into the other languages.

[8]We were only able to record code sizes for programs subsequently available to us at IC-Parc.

Strategy	ECL^{i}PS^e	Claire	OPL
Sol-O	897	-	716
Sol-E	713	-	651
Sol-C	1205	1290	-

Table 2: Code Size Comparison

The code sizes are quite similar, which suggests that all languages have a similar level of abstraction. The ECL^{i}PS^e code is somewhat longer than the OPL code, which can be explained by the fact that OPL is a special-purpose language where programs have a fixed structure and therefore certain things are implicit. The comparison with Claire shows even less deviation. Both are general high-level languages. Small differences are explained by coding style issues (e.g. use of auxiliary predicates/functions or not).

4.3.4 Results and Runtimes

Although the roster benchmark was selected with the explicit objective of avoiding the problems that are inherent in the comparison of large real-life applications, it proved difficult to exactly reproduce results. We used the finite-domain solvers on all three platforms, as well as built-in default labelling strategies to complement the explicitly programmed heuristics. Even though we sought to reproduce the search heuristics exactly, we ended up with quite different behaviours.

One reason is that the built-in labelling strategies of the platforms are subtly different. The other possible explanation is that heuristics which rely on evaluating the result of constraint propagation (domain sizes, etc.) are affected by different strengths of constraint propagation; i.e. low-level details of the solver implementation.

Due to these circumstances, the figures in the following tables must be interpreted carefully. When the platforms do not find the same solution, direct run time comparison is obviously meaningless. Even when a solution with the same cost is found, run times are not necessarily comparable: the solution might look completely different and may have been found along a different search path.

We look at three search strategies, Sol-E, Sol-O and Sol-C, as before. Each of them has been implemented on at least two of the platforms. The tables give the solutions and run times on the different platforms for a number of problem instances. The number in the name of the problem instance indicates the number of weeks for which a roster is calculated. If a program could find and prove an optimal solution within 60 seconds, the time needed is given, together with the optimal value of the objective function. If the program was aborted after 60 seconds because it had not been able to find and prove an optimal solution, then the value of the best solution it had found (if any) is given, along with (in parentheses) the time taken to find this solution.

	ECL^{i}PS^e	300 MHz	Claire	300 MHz	OPL	200 MHz
Instance	Cost	seconds	Cost	seconds	Cost	seconds
i5	1	0.15	1	0.09	1	0.28
i7	0	0.14	0	0.06	0	0.39
i9	0	0.39	1	(30.2)	0	0.77
i10	0	0.32	3	(0.10)	1	(0.38)
i12	0	1.95	1	(1.51)	1	(0.94)
i12a	1	(0.74)	6	(0.95)	0	38.1
i12b	1	(48.7)	-	(>60.0)	-	(>60.0)
i16	0	2.77	2	(0.34)	0	1.82
i18	0	3.31	6	(0.32)	6	(1.16)
i20	0	2.65	7	(1.13)	0	2.91
i21	0	11.8	-	(>60.0)	8	(0.71)
i23	4	(0.62)	4	(0.42)	4	(0.60)
i24	4	(3.45)	4	(1.39)	2	(6.64)
i26	9	(39.36)	-	(>60.0)	-	(>60.0)
i30	0	1.39	0	3.69	0	1.15

Table 3: Results for Sol-E search strategy

	ECL^{i}PS^e	300 MHz	Claire	300 MHz	OPL	200 MHz
Instance	Cost	seconds	Cost	seconds	Cost	seconds
i5	1	0.31			1	0.38
i7	2	(5.64)			0	65.41
i9	0	9.79			0	7.58
i10	4	(2.62)			4	(2.47)
i12	0	6.14			0	4.89
i12a	0	1.57			0	1.54
i12b	6	(32.92)			6	(26.09)
i16	1	(58.39)			0	40.26
i18	12	(6.59)			12	(4.01)
i20	7	(30.93)			7	(20.01)
i21	6	(22.40)			6	(18.13)
i23	21	(4.08)			21	(2.2)
i24	19	(5.86)			19	(1.82)
i26	26	(39.63)			24	(58.55)
i30	20	(51.19)			19	(58.1)

Table 4: Results for Sol-O search strategy

The ECL^{i}PS^e and Claire times were measured on a 300 MHz Pentium II machine. The OPL times were measured on a 200 MHz Pentium II machine.

In a number of cases (see table 3) ECL^{i}PS^e is the only platform to find solutions. In the cases where the solutions are the same, ECL^{i}PS^e is up to a

	ECL^{i}PS^e	300 MHz	Claire	300 MHz	OPL	200 MHz
Instance	Cost	seconds	Cost	seconds	Cost	seconds
i5	1	8.47	1	0.12		
i7	4	(2.78)	4	(0.91)		
i9	4	(3.14)	4	(1.32)		
i10	7	(1.86)	6	(57.9)		
i12	0	0.91	0	(8.20)		
i12a	5	(7.28)	5	(3.44)		
i12b	10	(1.63)	10	(1.12)		
i16	20	(56.06)	20	(28.3)		
i18	23	(24.41)	20	(42.2)		
i20	0	2.40	1	(2.61)		
i21	5	(1.99)	-	(>60.0)		
i23	32	(1.90)	32	(0.93)		
i24	23	(2.24)	23	(1.24)		
i26	30	(1.75)	-	(>60.0)		
i30	39	(1.91)	39	(0.72)		

Table 5: Results for Sol-C search strategy

factor of 2 slower than the other platforms. However, as can be seen from the tables, the differences between the strategies are much more important than the differences between the platforms.

The results support the view that the key to success is the richness and flexibility of the platform, rather than the performance of particular constraint solvers, or search primitives.

4.3.5 Review

The case study in this section represents a very unsatisfactory benchmarking exercise, in that the sample size is just one. Moreover the size of this section, i.e. the amount of text needed to report on the benchmark, is quite substantial, because the issues about development effort and program quality are complex. Therefore the benchmark may implicitly appear to have a greater weight than it should.

Of all the lessons that can be extracted from the benchmarking efforts reported in this paper, this may be the least "scientific" but it is very important. Benchmarking human effort is very costly, and therefore often uses small sample sizes - often reporting the experiences of just one user. On the other hand the issues involved are highly complex, and so the benchmark result may be accompanied by some discussion. The consequence is that the least reliable benchmarking exercise can easily take on a disproportionate weight in the overall benchmarking report. The lesson is that we must clearly recognise the relative significance of the different exercises within a benchmarking process. We must give weight to the results in proportion to their statistical and scientific validity.

5 Conclusion

The constraint logic programming benchmarks described in this paper fall into two main categories: *application benchmarks* and *unit tests*.

5.1 Application Benchmarks

The advantage of application benchmarks is that they appear to give a useful summary of the system being benchmarked. The results for an application benchmark are reported in a few figures. If the benchmark is being used to choose software for a planned application, it is hoped that the figures will predict the actual behaviour of the software on the application.

These summary figures are familiar to readers of consumer magazines which provide figures comparing products such as washing machines and cameras. The reader seeks an uncomplicated summary of the features and benefits of the alternative products so that they can be matched with his or her specific needs.

The risks of application benchmarks have been illustrated by some of the examples in this paper. It is easy to draw the wrong conclusion from the results of an application benchmark that does *not* correctly predict the behaviour of the software on the intended application.[9] This may be because the intended application has aspects that are not reflected in the benchmark application. On the other hand, the benchmark application may have aspects that are handled particularly well by the software but do not appear in the intended application.

The extra difficulty of comparing constraint programming platforms, as against washing machines for example, is that the platforms are general purpose. A washing machine has a relatively simple interface (a few buttons), and for each test there is a "right" way to use the machine.

By contrast a constraint programming platform can be used to solve a benchmark application in a wide variety of ways, one of which is to drop into some underlying programming or machine language. Thus the most efficient program for an application may not be a test of the platform at all. On the other hand, there may be an easy implementation on one platform, which has just the right built-in constraints, and a less easy implementation on another that is more run-time efficient, because it uses some application-specific code.

Choosing a constraint programming platform based on some application benchmarks is, therefore, deceptively simple. Drawing the right conclusion from such benchmarks is either an exercise in detective work or, more likely, a lottery.

5.2 Unit Tests

The advantage of unit tests is that it is clear what they measure. The drawback is that a software platform such as a constraint programming system has a very large number of facilities, which would require a huge number of unit tests.

[9] We recall a benchmark comparison in which a program using Constraint Handling Rules (CHRs) in ECL^{i}PS^e easily outperformed the other system. The reason eventually proved not to be the CHRs, but the good performance of large integer multiplication in ECL^{i}PS^e.

Choosing the right system by examining the results of all the unit tests would be an exhausting undertaking. It would also not necessarily be useful unless the intended application, and the programming features necessary to solve it, were fully known in advance.

It would not be enough to report that solver X did better on 58 unit tests and solver Y on only 42, and therefore solver X is the better. Clearly the weight given to each unit tests will depend on the purpose of the benchmarking. However it would be extraordinary for the user of the benchmark to have a precise and correct understanding of the appropriate weights.

For testing general purpose software platforms, such as database systems, a specified mix of functionalities has often been used in the past. The results are summarised in terms of the overall computational resources employed in getting through the whole set of benchmarks. This has the advantage of reducing the number of figures resulting from the benchmarking exercise, which is the main benefit of applications benchmarking.

On the other hand, in unit testing, the facilities being measured are clearly defined, and the importance given to each facility in the benchmarking is explicitly reflected in the number of uses of that facility included in the set of benchmarks. This clarity about what is being benchmarked is the main benefit of unit testing.

The assumption which makes this approach possible is that there is a fixed set of features and facilities to be tested, with fixed relative importance.

Such a set of benchmarks for constraint programming systems would be very useful. Unfortunately it is premature to ask for one at this time, because there is no fixed set of features and facilities, and certainly no understanding of their relative importance. It would, arguably, be possible to test finite domain constraint solvers in the different systems, if the finite domain technology had reached the level of maturity where a consensus on such a fixed prioritised set of features and facilities had been reached. This was the assumption underlying the benchmarking exercise of Fernandez and Hill.

However, their own benchmarking exercise explored features that were not, and are still not, standardised across the different finite domain solvers. One example, explicitly acknowledged by Fernandez and Hill, was *reified constraints*. Another example, which was not explicit, was an array multiplication constraint, variants of which were available in some of the systems at the time the benchmarking was carried out. This constraint would have strongly influenced the performance of the different systems on the *magic sequences* application benchmark.

5.3 The Future

Benchmarking constraint programming systems presupposes some standardisation in this area. Luckily the field is still a fertile area of research. As a consequence of this fertility standardisation is impractical. Moreover the attempt to impose standards at this point would be counter-productive.

The possibility of defining a standard for finite domain constraint programming appears to be more realistic. Such a standard would need to finesse the necessary run-time interfaces between finite domain solvers and other solvers that are not yet standardised.

However, even within the area of finite domain constraint solving, there is a major research initiative continuing. The emphasis now is on global constraints. As yet there is no claim for a fixed and final set of such global constraints. Rather, researchers are seeking an implementation infrastructure flexible and efficient enough to support the definition and implementation of future global constraints.

In fact the focus of our work on ECL^{i}PS^e is on hybrid algorithms for Large Scale Combinatorial Optimisation problems. We recognise the need for benchmark comparisons between systems. However because of the pace of research in this exciting and fast-moving area, benchmarking can only have secondary importance. When comparing CLP languages and systems, functionality, flexibility, orthogonality and in particular the support for communication between different solvers, are of much greater significance.

References

[BBV+97] A. Bachelu, P. Baptiste, C. Varnier, E. Boucher, and B. Legeard. Multi-criteria comparison between algorithmic, constraint logic and specific constraint programming on a real scheduling problem. In *Proc. Practical Application of Constraint Technology*, pages 47–63, 1997.

[BCD+00] BASF, CORE, Dash, Procter and Gamble, Lisbon University, Peugeot, Barbot, Loria, and COSYTEC. Large scale integrated supply chain optimisation software, 2000. `www.dash.co.uk/liscosweb/`.

[CHI99] Chic-2 home page. `www.icparc.ic.ac.uk/chic2/`, 1999.

[COC97] M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In *PLILP'97*, volume 1292 of *LNCS*, pages 191–206, 1997.

[Das01] Dash Optimization. XPRESS-MP. `www.dash.co.uk/`, 2001.

[DC93] D. Diaz and P. Codognet. A minimal extension of the wam for clp(FD). In *Proc. 10th International Conference on Logic Programming*, pages 774–790, 1993.

[EW00] H. El Sakkout and M. Wallace. Probe backtrack search for minimal perturbation in dynamic scheduling. *Constraints*, 5(4):359–388, 2000.

[EW01] A. Eremin and M. Wallace. Hybrid benders decomposition algorithms in constraint logic programming. In *Proc. CP'2001*, 2001.

[Fer97] A. Fernández. Self-referential quizzes. URL:
`www.lcc.uma.es/~afdez/srq/index.html`, 1997.

[FH00] A. Fernández and P. M. Hill. A comparative study of eight constraint programming languages over the Boolean and finite domains. *Journal of Constraints*, 5:275–301, 2000.

[Gra91] J. Gray, editor. *The Benchmark Handbook.* Data Management Systems. Morgan Kaufmann, 1991.

[Hay89] R. Haygood. A prolog benchmark suite for aquarius. Technical Report CSD-89-509, University of California, Berkeley, 1989.

[Hen96] Martin Henz. Don't be puzzled! In *Proceedings of the Workshop on Constraint Programming Applications, in conjunction with the Second International Conference on Principles and Practice of Constraint Programming (CP96)*, Cambridge, Massachusetts, USA, August 1996.

[HEW+98] M. Hajian, H. El-Sakkout, M. Wallace, J. Lever, and E. B. Richards. Towards a closer integration of finite domain propagation and simplex-based methods. *Annals of Operations Research*, 81:421–431, 1998.

[ILO01] ILOG. CPLEX. `www.ilog.com/products/cplex/`, 2001.

[KV98] B. Kernighan and C. Van Wyk. Timing trials, or the trials of timing: Experiments with scripting and user-interface languages. *Software Practise and Experience*, 28(8):819–843, 1998. `cm.bell-labs.com/cm/cs/who/bwk/interps/pap.html`.

[McC99] D. McClain. A comparison of programming languages for scientific processing. URL:
`www.azstarnet.com/~dmcclain/LanguageStudy.html`, 1999.

[MF85] A. K. Mackworth and E. C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25(1):65–74, 1985.

[MW96] T. Müller and J. Würtz. Interfacing propagators with a concurrent constraint language. In *PLILP'96*, pages 195–206, 1996.

[Nad89] B. Nadel. Constraint satisfaction algorithms. *Computational Intelligence*, 5:188–224, 1989.

[OAI+98] Ovako, Air Liquide, IQSOFT, PrologIA, and SICS. Trial application using constraint programming in industrial manufacturing, 1998. ESPRIT Project No 23365, URL:
`www.sics.se/col/projects/tacit/description.html`.

[Per87] F. Pereira. AI expert prolog benchmarks, 1987. URL: `www.icparc.ic.ac.uk/~mgw/aiexpertbench.txt`.

[Ref99] P. Refalo. Tight cooperation and its application in piecewise linear optimization. In *CP'99*, volume 1713 of *LNCS*, pages 373–389, 1999.

[RW98a] R. Rodosek and M. Wallace. A generic model and hybrid algorithm for hoist scheduling problems. In *Proc. Principles and Practice of Constraint Programming*, pages 385–399, 1998.

[RW98b] R. Rodosek and M.G. Wallace. A generic model and hybrid algorithm for hoist scheduling problems. In *Proceedings of the 4th International Conference on Principles and Practic e of Constaint Programming*, pages 385–399, Pisa, 1998.

[RWH99] R. Rodosek, M. Wallace, and M. Hajian. A new approach to integrating mixed integer programming with constraint logic programming. *Annals of Operations Research*, 86:63–87, 1999.

[Van89] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* Logic Programming. MIT Press, 1989.

[Wac97] K. Waclena. My programming language crisis. URL: `www.lib.uchicago.edu/keith/crisis/`, 1997.

[Wil93] H. P. Williams. *Model Building in Mathematical Programming.* John Wiley and Sons, 1993.

[WNS97] M. Wallace, S. Novello, and J. Schimpf. Eclipse - a platform for constraint programming. *ICL Systems Journal*, 12(1):159–200, 1997.

[WS02] M. Wallace and J. Schimpf. Finding the right hybrid algorithm - a combinatorial meta-problem. *Annals of Mathematics and Artificial Intelligence*, Special issue on Large Scale Combinatorial Optimisation and Constraints, 2002. To appear.

[Zho98] N.-F. Zhou. A high-level intermediate language and the algorithms for compiling finite-domain constraints. In *Proc. Joint International Conference on Logic Programming*, pages 70–84. MIT Press, 1998.