Software Tools Supporting Integration

Tallys Yunes

Abstract This chapter provides a brief survey of existing software tools that enable, facilitate and/or support the integration of different optimization techniques. We focus on tools that have achieved a reasonable level of maturity and whose published results have demonstrated their effectiveness. The description of each tool is not intended to be comprehensive. We include references and links to detailed accounts of each tool for the interested reader, and we recommend that the reader consult their developers and/or vendors for the latest information about upgrades and improvements. Our purpose is to summarize the main features of each tool, highlighting what it can (or cannot) do, given the current version at the time of writing. We conclude the chapter with suggestions for future research directions.

1 Introduction

This chapter provides a brief survey of existing software tools that enable, facilitate and/or support the integration of different optimization techniques. By *optimization techniques* we mean linear programming (LP), mixed-integer linear programming (MILP), non-linear programming (NLP), constraint programming (CP), local search (LS), large neighborhood search (LNS), and their derivatives. Integration can occur in many ways and at different levels. It ranges from a simple information exchange between two solvers implementing two versions of the same mathematical model, to a full-fledged synchronization of multiple methods that interact closely during an iterative search procedure. Decomposition methods such as branch-and-price (B&P) [12] and Benders decomposition [14] also lend themselves to effective integrated approaches and are, therefore, also covered here. Due to space limitations, we were not able to include a comprehensive list of all available tools. We focus, however, on

Tallys Yunes
Department of Management Science, School of Business Administration, University of Miami, Coral Gables, FL 33124-8237, e-mail: tallys@miami.edu

those tools that have achieved a reasonable level of maturity and whose published results have demonstrated their effectiveness.

The tools typically come in two main categories: *high level* and *low level*. High-level tools are those that offer a high-level modeling language to the user and do not necessarily require computer programming at the level of languages such as C, C++, Java or Prolog. Low-level tools are those that require the user to write low-level code to be able to take advantage of their integration abilities. This is usually done in one of two ways: writing code that includes calls to a library of functions or object classes (the tool's API), or writing code in a computer programming language that has been enriched with new declarations and commands that provide access to the underlying algorithms and solvers (e.g. extensions to the Prolog language). We decided, however, not to separate the tools into the above two groups. Because of the rapidly evolving nature of optimization software and the ongoing introduction of new features and interfaces, such a classification would soon be obsolete. Moreover, some tools are actually a mixture of both high and low level features. Therefore, we have opted to list the tools in plain alphabetical order.

The description of each tool focuses on its defining characteristics, and does not attempt to list all its facilities. We include references and links to detailed accounts of each software for the interested reader, and we recommend that the reader consult the developers and/or vendors of each tool for the latest information about upgrades and improvements. Our purpose is to summarize the main features of each tool, highlighting what it can (or cannot) do, given the current version at the time of writing. The contents of the following sections have been taken from published material about each tool, such as: user manuals, web sites, books, and research papers. We have tried, to the extent possible, to stay true to the software descriptions provided by the respective authors and developers. Changes to those descriptions have been kept to a minimum, and were made with the sole intent of improving clarity and removing biases and subjective statements. Finally, the code excerpts presented in this chapter are for illustration purposes only and do not intend to represent the best way of modeling any particular problem.

This chapter is organized as follows. Section 2 presents the main features that are currently available to facilitate the integration of optimization techniques. Descriptions of each tool appear in sections 3 through 11. For each tool, excluding those in Sect. 11, we include a *data sheet*, which is a table summarizing some of its characteristics for easy reference, such as: supported features, solvers and platforms, details on availability, references and additional notes. When available, developers' names are listed in alphabetical order. Some tools also include a "coming soon" paragraph with new features that are currently under development. Section 12 concludes the chapter with final remarks and directions for future research.

2 Existing Features that Facilitate Integration

In this section we provide a classification scheme for the different kinds of features and functionalities that can appear in a software system supporting the integration of different optimization techniques. The purpose of this classification is to create an organized hierarchy of features that can be easily referenced in the subsequent sections.

1. *User interface*: how the user interacts with (i.e. furnishes a model to) the tool.
   a. Graphical interface/IDE available (assumes the tool has a high-level modeling language).
   b. Tool can read an input file with model description written in a high-level language.
   c. User writes code to use the tool's API.
2. *Solver support*: kinds of solvers that are part of (or can interface with) the tool.
   a. LP, MILP
   b. NLP
   c. CP
   d. SAT
3. *Relaxation support*: kinds of model relaxations that are supported by the tool.
   a. Linear
   b. Non-linear
   c. Lagrangian
   d. Domain store (from a CP solver)
4. *Inference mechanisms*: forms of inference supported by the tool.
   a. Pre-processing (upfront elimination/tightening of variables and constraints).
   b. Cutting planes.
   c. Domain filtering/reduction.
   d. Logical resolution mechanisms (as in satisfiability problems).
5. *Search mechanisms*: forms of search supported by the tool.
   a. Complete branching (e.g. tree search, branch-and-bound).
   b. Incomplete branching (e.g. LDS).
   c. Simple local search (no meta-heuristic).
   d. Meta-heuristics (e.g. tabu search, simulated annealing).
6. *Decomposition mechanisms*: forms of decomposition supported by the tool.
   a. Dantzig-Wolfe.
   b. Column generation/Branch-and-Price (B&P) (master problem and pricing problem can potentially be solved by different techniques).
   c. Benders.

7. *Search control*: how much control the user has over the search mechanisms.
   a. High-level (via modeling language), declarative (i.e. fixed forms of control).
   b. High-level, imperative (detailed and flexible control).
   c. Low-level (coding through an API).

We will refer back to this classification scheme in the data sheet of each tool. For instance, to indicate that a certain tool has an IDE, interfaces with a SAT solver and supports Benders decomposition, we use the codes 1a, 2d, and 6c.

The classification scheme above refers to the support for integration only. Hence, if a given tool accepts LP files as input (a kind of high-level input), but requires low-level coding for access to its integration features, its user interface will be classified as 1c, rather than 1b.

3 BARON

BARON [57] was developed with the main purpose to find global solutions to non-linear and mixed-integer non-linear programs. It is based on the idea of *branch-and-reduce*, which can be seen as a modification of branch-and-bound in which a variety of logical inferences made during the search tree exploration help reduce variable domains and tighten the problem relaxation. Two of the main domain reduction techniques used by BARON are:

Optimality-based range reduction: Let $x_j - x_j^U \leq 0$ be a constraint that is active at the optimal solution of the relaxation of the current search node (i.e. variable x_j is at its upper bound), and let $\lambda_j > 0$ be the corresponding Lagrange multiplier. If we know an upper bound U and a lower bound L on the value of the optimal solution, a valid lower bound for x_j can be calculated as $x_j^U - (U-L)/\lambda_j$. (An analogous procedure can be used to calculate an improved upper bound for x_j.). If variable x_j is at neither of its bounds in the solution of the relaxation, we can *probe* its bounds (temporarily fix x_j at its bounds, and resolve the relaxation) to obtain possibly better bounds. This technique extends to arbitrary constraints of the type $g_i(x) \leq 0$.

Feasibility-based range reduction: This is a process that generates constraints that cut off infeasible parts of the search space. For instance, given a constraint $\sum_{j=1}^{n} a_{ij}x_j \leq b_i$, one of the constraints below is valid for a pair (i,h) with $a_{ih} \neq 0$:

$$x_h \leq \frac{1}{a_{ih}} \left(b_i - \sum_{j \neq h} \min \left\{ a_{ij}x_j^U, a_{ij}x_j^L \right\} \right), \quad a_{ih} > 0 \tag{1}$$

$$x_h \geq \frac{1}{a_{ih}} \left(b_i - \sum_{j \neq h} \min \left\{ a_{ij}x_j^U, a_{ij}x_j^L \right\} \right), \quad a_{ih} < 0 \tag{2}$$

The above inequalities can be interpreted as an approximation to the optimal bound tightening procedure that involves solving the $2n$ linear programs

$$\left\{ \min \pm x_k \text{ s.t.} \sum_{j=1}^{n} a_{ij} x_j \leq b_i,\ i = 1, \ldots, m \right\},\ \ k = 1, \ldots, n \tag{3}$$

BARON makes extensive use of (1) and (2) throughout the search, along with a limited use of (3), mostly at the root node, for selected variables. Non-linear analogues of (1) and (2) are also utilized throughout the search.

For a given problem, BARON's global optimization strategy integrates conventional branch-and-bound with a wide variety of range reduction tests. These tests are applied to every subproblem of the search tree in pre- and post-processing steps to contract the search space and reduce the relaxation gap. Many of the reduction tests are based on duality and are applied when the relaxation is convex and solved by an algorithm that provides the dual, in addition to the primal, solution of the relaxed problem. Another crucial component of the software is the implementation of heuristic techniques for the approximate solution of optimization problems that yield improved bounds for the problem variables. Finally, the algorithm incorporates a number of compound branching schemes that accelerate convergence of standard branching strategies.

Typically, the relaxed problem is constructed using factorable programming techniques (see [42, 43]), so that the relaxations are exact at the variable bounds. Therefore, the tightness of the relaxation depends on the tightness of the variable bounds. BARON implements the construction of underestimators for many non-convex terms such as: bi-linearities ($x_i x_j$), linear fractional terms (x_i/x_j), univariate terms such as x^n with n odd, etc. Note that outer approximations do not necessarily need to be linear.

BARON comes in the form of a callable library. The software has a core component which can be used to solve any global optimization problem for which the user supplies problem specific subroutines, primarily for lower and upper bounding. In this way, the core system is capable of solving very general problems. In addition to the general purpose core, the BARON library also provides ready to use specialized modules covering several classes of problems. These modules work in conjunction with the core component and require no coding on the part of the user. Some of the available problem classes are: separable concave quadratic programming, separable concave programming, problems with power economies of scale (Cobb-Douglas functions), fixed-charge programming, fractional programming, univariate polynomial programming, linear and general linear multiplicative programming, indefinite quadratic programming, mixed integer linear programming, and mixed integer semidefinite programming.

BARON includes a factorable non-linear programming module, which is the most general of the supplied modules that do not require any coding from the user. The input to this module is through BARON's parser and it can solve fairly general non-linear programs where the objective and constraints are factorable functions (i.e. recursive compositions of sums and products of functions of single variables). Most functions of several variables used in non-linear optimization are factorable

and can be easily brought into separable form. The types of functions currently allowed in this module include $\exp(x)$, $\ln(x)$, x^α for $\alpha \in \mathbb{R}$, and β^x for $\beta \in \mathbb{R}$.

The theoretical and empirical work that is embodied in BARON earned Nikolaos Sahinidis and Mohit Tawarmalani the 2004 INFORMS Computing Society Prize and the 2006 Beale-Orchard-Hays Prize from the Mathematical Programming Society. BARON is customarily used as a benchmark code when it comes to solving global optimization problems.

Table 1 BARON Data Sheet

Feature	Description
Tool name	BARON: Branch-and-Reduce Optimization Navigator
User interface	1b, 1c
Solver support	2a, 2b
Relaxation support	3a, 3b
Inference mechanisms	4a, 4b, 4c
Search mechanisms	5a, 5c
Decomposition mechanisms	N/A
Search control	7a + some added flexibility through GAMS, 7c
Written in	C and Fortran
Supported platforms	Linux, Windows, AIX
Main developers	Nikolaos Sahinidis and Mohit Tawarmalani
Availability	Available under the AIMMS and GAMS modeling languages. See http://archimedes.cheme.cmu.edu/baron/baron.html.
References	[57]
Notes	Also available on the NEOS server.

Most of BARON's specialized modules require the use of an LP solver. An NLP solver is optional, but frequently beneficial. The current release (8.1) can work with many different LP, NLP and SDP solvers such as OSL, CPLEX, CONOPT, MINOS, SDPA, and SNOPT.

Coming soon to BARON: The ability to solve MILP relaxations for problems that involve a combinatorial component.

4 Comet

One of the main innovations of Comet [59] is Constraint-based Local Search (CBLS), a computational paradigm based on the idea of specifying local search algorithms as two components: a high-level model describing the applications in terms of constraints, constraint combinators, and objective functions, and a search procedure expressed in terms of the model at a high level of abstraction. CBLS makes it possible to build local search algorithms compositionally, to separate modeling from search, to promote reusability across many applications, and to exploit

problem structure to achieve high performance. The computational model underlying CBLS uses constraints and objectives to drive the search procedure toward feasible or high-quality solutions. Constraints incrementally maintain their violations, and objectives maintain their values. Moreover, constraints and objectives are differentiable and can be queried to evaluate the effect of moves on their violations and values.

Comet is an object-oriented language featuring both a constraint-based local search engine and a constraint-programming solver. Comet models for these two paradigms are essentially similar and are expressed in modeling languages featuring logical and cardinality constraints, global/combinatorial constraints, numerical (possibly non-linear) constraints, and vertical extensions for specific application areas (e.g. scheduling abstractions). The search components are also expressed in a rich language for controlling graph- and tree- search explorations. The constraint-programming solver also views combinatorial optimization as the combination of model and search components. It provides the traditional expressiveness of constraint programming with a language for expressing search algorithms.

Comet also features high-level abstractions for parallel and distributed computing, based on parallel loops, interruptions, and work stealing. In addition, it provides a declarative language for specifying model-driven visualizations of various aspects of an optimization algorithm. Finally, being an open language, Comet allows programmers to add their own constraints and objectives, as well as their own control abstractions.

To illustrate some of Comet's modeling constructs, let us consider the well-known n-queens problem. We are given an $n \times n$ chessboard. The objective is to place n queens on the chessboard so that no two queens can attack each other (i.e. no two queens lie on the same row, column or diagonal). Figure 1 shows a Comet model for the n-queens problem using local search. The algorithm is based on the min-conflict heuristic [45] but uses a greedy heuristic to select the variables. Because no two queens can be placed on the same column, we can assume that the i-th queen is placed on the i-th column, and variable `queen[i]` represents the row assigned to the i-th queen. We start by including the local solver library and the local solver (lines 01 and 02). Line 06 declares the incremental variables (`queen`) and performs an initial random assignment of queens to rows. The three alldifferent constraints in lines 09–11 check for violations of the row and diagonal constraints. Then, at each iteration, we choose the queen violating the most constraints (line 16) and move it to a position (row) that minimizes its violations (line 17). These steps are repeated until a feasible solution is found (the constraint system `S` has no violations) or the number of iterations becomes greater than $50n$ (line 15). The expression `S.getAssignDelta(queen[q],v)` queries the constraint system to find out the impact of assigning value `v` to queen `q`. This query is performed in constant time because of the invariants maintained in each of the constraints. The assignment `queen[q] := v` in line 18 induces a propagation step that recomputes the violations of all constraints.

More recently, Comet has added support for CP, LP and MIP solvers, as well as the ability to do column (variable) generation during search. To access the above

```
01. import cotls;
02. Solver<LS> m();
03. int n = 16;
04. range Size = 1..n;
05. UniformDistribution distr(Size);
06. var{int} queen[Size](m,Size) := distr.get();
07.
08. ConstraintSystem<LS> S(m);
09. S.post(alldifferent(queen));
10. S.post(alldifferent(all(i in Size) queen[i] + i));
11. S.post(alldifferent(all(i in Size) queen[i] - i));
12. m.close();
13.
14. int iter = 0;
15. while (S.violations() > 0 && iter < 50*n) {
16.    selectMax(q in Size)(S.violations(queen[q]))
17.       selectMin(v in Size)(S.getAssignDelta(queen[q],v))
18.          queen[q] := v;
19.    iter++;
20. }
```

Fig. 1 Comet model for the *n*-queens problem

solvers, the user can write statements like `Solver<CP> m();`, `Solver<LP> m();` or `Solver<MIP> m();` instead of `Solver<LS>`. Variable creation and constraint posting are modified accordingly. To create finite domain CP variables for the *n*-queens problem, we can write `var<CP>int queen[Size](m,Size)`.

Table 2 Comet Data Sheet

Feature	Description
Tool name	Comet
User interface	1b, 1c
Solver support	2a, 2c
Relaxation support	3a, 3d
Inference mechanisms	4a, 4c
Search mechanisms	5a, 5b, 5c, 5d
Decomposition mechanisms	6b
Search control	7a, 7b, 7c
Written in	C++ and Assembly
Supported platforms	Linux, Mac OS X, Windows XP
Developers	Laurent Michel and Pascal Van Hentenryck
Availability	Free for non-commercial use at http://www.dynadec.com. See also http://www.comet-online.org.
References	[59]
Notes	Also includes an ODBC module to grant access to databases.

The linear solvers supported by the current Comet release (1.2) are LP_SOLVE and the COIN-OR LP solver CLP.

Coming soon to Comet: A uniform GUI/Visualization tool and CP set variables.

5 ECL^{i}PS^e

ECL^{i}PS^e [61, 7] is a Prolog-based system whose aim is to serve as a platform for integrating various Logic Programming extensions, in particular Constraint Logic Programming (CLP). The kernel of ECL^{i}PS^e is an efficient implementation of standard (Edinburgh-like) Prolog as described in basic Prolog texts [20]. It is built around an incremental compiler which compiles the ECL^{i}PS^e source code into WAM-like code [62], and an emulator of this abstract code. The ECL^{i}PS^e logic programming system was originally an integration of ECRC's SEPIA [44], Megalog [16] and parts of the CHIP [24] systems. It was then further developed into a Constraint Logic Programming system with a focus on hybrid problem solving and solver integration. ECL^{i}PS^e is now an open-source project, with the support of Cisco Systems.

There are two ways of running ECL^{i}PS^e programs. The first is using an interactive graphical user interface to the ECL^{i}PS^e compiler and system, which is called *tkeclipse*. The second is through a more traditional command-line interface. The graphical user interface provides many useful features such as a tracer (for debugging), a scratch-pad (to experiment with small pieces of code), statistics (e.g. CPU and memory usage), predicate library help, etc.

ECL^{i}PS^e has introduced many specific language constructs to overcome some of the main deficiencies of Prolog, such as: the ability to use named structures with field names, loop/iterator constructs, a multidimensional array notation, pattern matching for clauses, soft cuts, a string data type, etc.

ECL^{i}PS^e allows the use of different constraint solvers in combination. The different solvers may share variables and constraints. This can be done by combining the `eplex` and `ic` solver libraries of ECL^{i}PS^e. The `eplex` library gives access to LP and MIP solvers (linear numeric constraints and integrality constraints), and the `ic` library implements typical constraint propagation algorithms (finite domain constraints and interval constraints). In a hybrid model, the `ic` solver communicates new tightened bounds to the `eplex` solver. These tightened bounds have typically been deduced from non-linear constraints (e.g. global constraints) and thus the linear solver benefits from information which would not otherwise have been available to it. On the other hand, the `eplex` solver often detects inconsistencies which would not have been detected by the `ic` solver. Moreover, it returns a bound on the optimization function that can be used by the `ic` constraints. Finally, the optimal solution returned by `eplex` to the "relaxed" problem comprising just the linear constraints can be used in a search heuristic that can focus the `ic` solver on the most promising parts of the search space. Other useful libraries are `colgen` (support for

column generation) and `repair` (helps propagate solutions generated by a linear solver to other variables handled by the domain solver).

To illustrate the ECL^{i}PS^e syntax, let us consider a simple example. We are given a time instant t, and three tasks such that the running time of tasks 1 and 2 are, respectively, 3 and 5 time units. We want to find start times for each task such that two constraints are satisfied: exactly one of tasks 1 and 2 is running at time t, and both tasks 1 and 2 precede task 3. The corresponding ECL^{i}PS^e code appears in Fig. 2. Line 01 loads the `ic` and `eplex` libraries. Line 02 defines a predicate that

```
01. :- lib(ic), lib(eplex).
02. overlap(S,D,T,B) :- ic:(B #= ((T $>= S)and(T $=< S+D-1))).
03. hybrid(Time,[S1,S2,S3],Obj) :-
04.     ic:(Obj $:: -1.0Inf..1.0Inf),
05.     ic:([S1,S2,S3]::1..20),
06.     overlap(S1,3,Time,B1), overlap(S2,5,Time,B2),
07.     ic:(B1+B2 #= 1),
08.     eplex:(S1+3 $=< S3), eplex:(S2+5 $=< S3),
09.     eplex:eplex_solver_setup(min(S3),Obj,
            [sync_bounds(yes)],[ic:min,ic:max]),
10.     labeling([B1,B2,S1,S2]).
```

Fig. 2 ECL^{i}PS^e model illustrating solver integration

associates a boolean variable `B` with whether a task with start time `S` and duration `D` is running at time `T`. Lines 04 and 05 give initial domains to the `Obj` variable, which stores objective function bounds, and to the start time variables `S1`, `S2`, `S3`. To enforce the first constraint, we state in line 07 that exactly one of the boolean variables `B1` and `B2` has to be equal to 1. Note that the prefix `ic:` indicates that all of the above constraints are posted to the `ic` solver. We ask the `eplex` solver to handle the second constraint in line 08, while line 09 sets up the objective (minimize `S3`), and passes `ic` bounds to the linear solver before the problem is solved (`[sync_bounds(yes)]`). The statement `[ic:min,ic:max]` triggers the LP solver in case of bound changes on the `ic` side. We search for a solution by labeling variables in line 10. To send a constraint, say $x+2 \geq y$, to both the `eplex` and `ic` solvers at once, we could write `[eplex,ic]:(X + 2 $>= Y)`.

Three decomposition techniques that are amenable to hybridization are column generation, Benders decomposition, and Lagrangian relaxation. All three have been implemented in ECL^{i}PS^e and used to solve large problems.

The current release of ECL^{i}PS^e (6.0) can interface with the following LP/MIP solvers: CPLEX, Xpress-MP (versions developed until 2005), and the COIN-OR LP solver CLP.

Coming soon to ECL^{i}PS^e: Integration with the Gecode [56] constraint solver library; improved propagator support; new and more efficient global constraints such as cardinality, 2-D alldifferent, sequence, etc.; search annotations and search tree

Table 3 ECL^{i}PS^e Data Sheet

Feature	Description
Tool name	ECL^{i}PS^e
User interface	1a (Saros prototype under development), 1b, 1c
Solver support	2a, 2c
Relaxation support	3a, 3c, 3d
Inference mechanisms	4c, ?
Search mechanisms	5a, 5b, 5c, 5d
Decomposition mechanisms	6b, 6c
Search control	7a, 7b, 7c
Written in	ECL^{i}PS^e/Prolog
Supported platforms	Linux, Mac OS X, Solaris, Windows
Developers	See list at http://sourceforge.net/projects/eclipse-clp
Availability	Freely available at http://www.eclipse-clp.org
References	[61, 7]

display. See http://www.eclipse-clp.org/reports/roadmap.html for a road map of future ECL^{i}PS^e releases.

6 G12

The G12 project [55] started by National ICT Australia (NICTA) seeks to develop a software platform for solving large-scale industrial combinatorial optimization problems. The core design involves three languages: Zinc, Cadmium and Mercury (group 12 of the periodic table). Zinc [28, 13] is a declarative modeling language for expressing problems independently of any solving methodology. Cadmium [25] is a mapping language for mapping Zinc models to underlying solvers and/or search strategies, including hybrid approaches. Finally, the existing Mercury language [54] will be extended as a language for building extensible and hybridizable solvers. The same Zinc model, used with different Cadmium mappings, allows the user to experiment with different complete, local, or hybrid search approaches for the same problem. Cadmium mappings also enable the application of pre-processing steps (e.g. bounds tightening) and transformation steps (e.g. linearization of subproblems) to the original Zinc model. Some of the features provided by Zinc are: mathematical notation-like syntax (overloading, iterations, sets, arrays), expressive constraints, support for different kinds of problems (including preferences, i.e. soft constraints), interface to hybrid solvers (like lazy FD), separation of data from model, high-level data structures and data encapsulation, extensibility (user defined functions), reliability (type checking, assertions), and *annotations* which allow non-declarative information (such as search strategies) and solver-specific information (such as variable representations) to be layered on top of the declarative model. Another G12 component is MiniZinc [47], which is a subset of Zinc that provides many modeling capabilities while being easier to implement. The target language of MiniZinc

is FlatZinc, a low-level solver input language that offers an easier way for solver writers to provide reasonable MiniZinc support.

To illustrate the Zinc language, let us consider the following trucking example from [50]. We are given `T` trucks, each having a cost `Cost[t]` and an amount of material it can load `Load[t]`. We are also given `P` time periods. In each time period `p`, a given demand of material (`Dem[p]`) has to be shipped. Each truck `t` also has constraints on usage: in each consecutive `K[t]` time periods, it must be used at least `L[t]` and at most `U[t]` times. The Zinc model of this problem appears in Fig. 3. In

```
01. int: P;                          type Per = 1..P;
02. int: T;                          type Trucks = 1..T;
03. array[Per]    of int: Dem;       array[Trucks] of int: Cost;
04. array[Trucks] of int: Load;      array[Trucks] of int: K;
05. array[Trucks] of int: L;         array[Trucks] of int: U;
06. array[Per]    of var set of Trucks: x;

07. constraint forall(p in Per)(
      sum_set(x[p],Load) >= Dem[p]);
08. constraint forall(t in Trucks)(
      sequence([bool2int(t in x[p])|p in Per],L[t],U[t],K[t]));

09. solve minimize sum(p in Per)(sum_set(x[p], Cost));
```

Fig. 3 Zinc model for the trucking

each time period `p`, we need to choose which trucks to use in order to ship enough material and satisfy the usage limits. Hence, variable `x[p]` represents the subset of trucks used in period `p`. In line 07, the `sum_set`(S,f) function returns $\sum_{e\in S} f(e)$; in line 08, `sequence`$([y_1,\ldots,y_n],\ell,u,k)$ constrains the sum of each subsequence of y variables of length k to be between ℓ and u. As it stands, this model is directly executable in a finite domain solver which supports set variables. In Zinc, we can control the search by adding an annotation on the `solve` statement. For example,

```
solve :: set_search(x,"first_fail","indomain","complete")
   minimize sum(p in Per)(sum_set(x[p], Cost));
```

indicates that we label the `x` variables with smallest domain first (`first_fail`) by first trying to exclude an unknown element of the set and then including it (`indomain`) in a complete search.

To use Dantzig-Wolfe decomposition and column generation on the trucking model, we need to annotate the model to explain what parts define the sub-problems, which solver is to be used for each subproblem, and which solver is to be used for the master problem. The annotations would look as follows

```
array[Per] of var set of Trucks: x :: colgen_var;

constraint forall(p in Per)(
sum_set(x[p],Load) >= Dem[p] ::
        colgen_subproblem_constraint(p,"mip"));
```

```
solve :: colgen_solver("lp") :: lp_bb(x, most_frac, std_split)
minimize sum(p in Per)(sum_set(x[p], Cost));
```

which means that variables `x` will be used in column generation. The subproblems numbered 1 through `P` are defined in terms of their constraints and their solver (`mip`). Finally, the solver for the master problem and the search specification, branch-and-bound selecting the most fractional variable first and performing a standard split, are attached to the `solve` statement. Since column generation is to be used, the transformation must linearize the master constraints and objective function. This can be done in Zinc by giving linear definitions to the `sum_set` and `sequence` constraints. For the complete details, see [50].

The G12 facilities have been successfully used for a variety of computational experiments involving problems such as radiation, cumulative scheduling and hoist scheduling. Another hybrid algorithm supported by G12 is *Lazy-FD*, which is a tight integration of FD and SAT that has shown consistently good performance on FD problems [48].

Table 4 G12 Data Sheet

Feature	Description
Tool name	The G12 Project
User interface	1b, 1c
Solver support	2a, 2c, 2d
Relaxation support	3a, 3d
Inference mechanisms	4a, 4b, 4c, 4d
Search mechanisms	5a, 5b, 5c, 5d
Decomposition mechanisms	6a, 6b
Search control	7a, 7b, 7c
Written in	Mercury and C++
Supported platforms	Linux, Mac OS X, Windows
Developers	See list at http://www.nicta.com.au/research/projects/ /constraint_programming_platform (full URL was split in two lines)
Availability	See http://www.g12.csse.unimelb.edu.au
References	[13, 28, 47, 50, 55]

Zinc/MiniZinc (as of version 0.9) has been interfaced to a number of finite domain solvers (including Gecode [56] and ECL$^{\text{i}}$PS$^{\text{e}}$ [61]), LP solvers (including CPLEX, GLPK and COIN-OR CLP through COIN-OR OSI), and SAT solvers (including MiniSAT and TiniSAT).

Coming soon to G12: Propagation-based solving where individual constraints can be annotated with the solver where they should be sent, for example, `x >= y :: lp :: fd` means that the constraint `x >= y` is sent to both the LP and finite domain solvers; and annotation of models to split the problem into two parts: this-then-that, where the first part is solved and used as input to the second part.

7 ILOG CP Optimizer and OPL Development Studio

The ILOG OPL Development Studio [58] (OPL for short) is designed to support ILOG CPLEX as well as ILOG CP Optimizer [37, 36]. OPL allows users to develop single models in either technology or multi-model solutions that use either or both technologies combined. A scripting language (ILOG Script for OPL) provides an additional level of control, enabling the user to create algorithms that run multiple models sequentially or iteratively, and to exchange information among them. OPL also includes an Integrated Development Environment (IDE), which is a graphical user interface providing all the typical development facilities and support, including: searching for conflicts between constraints in infeasible mathematical programming (MP) models, visualizing the state of variables at some point during the search for a solution, connecting to a database or to a spreadsheet to read and write data, debugging, etc. The current OPL version as of this writing is 6.0.1.

Since version 2.0, the ILOG CP Optimizer provides a new scheduling language supported by an automatic search that is meant to be robust and efficient (i.e. its default behavior, without much user intervention or fine tuning, should perform well on average). This new scheduling model was designed with the requirement that it should, among other things, be accessible to mathematical programmers, be simple and non-redundant, fit naturally into a CP paradigm, and be expressive enough to handle complex applications. The scheduling language is available in C++, Java, and C#, as well as in OPL itself. The automatic search is based on a Self-Adapting Large Neighborhood Search that iteratively *unfreezes* and *re-optimizes* a selected fragment of the current solution. ILOG CP Optimizer introduces a conditional interval formalism that extends classical constraint programming by introducing additional mathematical concepts (such as intervals, sequences or functions) as new variables or expressions to capture the temporal aspects of scheduling For example:

Interval Variables: An interval variable a is a decision variable whose domain $\mathrm{dom}(a)$ is a subset of $\{\bot\} \cup \{[s,e) \mid s,e \in \mathbb{Z},\ s \leq e\}$. If $\mathrm{dom}(a) = \bot$, the interval is said to be *absent* (*present* otherwise). An absent interval variable is not considered by any constraint or expression on interval variables in which it is involved. To model the basic structures of scheduling problems, a number of special constraints on interval variables exist, such as precedence constraints, and time spanning constraints.

Sequence Variables: Sequence variables are a type of decision variable whose value is a permutation of a set of interval variables. They are inspired by problems that involve scheduling a set of activities on a disjunctive resource. Constraints on sequence variables are provided for ruling out illegal permutations (sequencing constraints) or for stating a particular relation between the order of intervals in the permutation and the relative position of their start and end values (no-overlap constraints).

Cumul Function Expressions: Cumul Function Expressions are expressions that are used to represent the usage of a cumulative resource over time. This usage is

represented as a sum of individual contributions of intervals (some time intervals make the resource consumption go up, others make it go down).

In a recent study of three problems from the scheduling literature, compact models written in OPL for CP Optimizer produced results that, on average, outperformed state-of-the-art problem-specific approaches for those problems (see [36] for details).

To illustrate the integration capabilities of OPL, CPLEX and CP Optimizer, we will briefly go over the steps to solve a configuration problem using column generation. We will use the CP Optimizer engine to generate new possible configurations, and the CPLEX engine to solve the problem of selecting the best combination of configurations. This configuration problem involves placing objects of different materials (glass, plastic, steel, wood, and copper) into bins of various types (red, blue, green), subject to capacity (each bin type has a maximum) and compatibility constraints (e.g. red bins cannot contain plastic or steel). All objects must be placed into a bin and the total number of bins must be minimized. The idea is to write a model file that uses CP to *generate* bin configurations, a second model file that uses CPLEX to *select* a subset of configurations, and a script file that coordinates the execution of the previous two and passes information from one to the other. The generation model (`generate.mod`) has a color variable to indicate the bin color and object variables indicating how many objects of each material are included in the bin. It then writes the compatibility constraints as logical expressions and imposes the bin capacity constraints. The selection model (`select.mod`) has one variable for each configuration created as input and an objective that minimizes the number of bins produced. The only constraints are to satisfy the demand for each material. The script model appears in Fig. 4. Line 01 associates the internal name `genBin` to

```
01. var genBin = new IloOplRunConfiguration("generate.mod");
02. genBin.oplModel.addDataSource(data);
03. genBin.oplModel.generate();
04. genBin.cp.startNewSearch();
05. while (genBin.cp.next()) {
06.   genBin.oplModel.postProcess();
07.   data.Bins.add(genBin.oplModel.newId,
                    genBin.oplModel.colorStringValue,
                    genBin.oplModel.n.solutionValue);
08. }
09. genBin.cp.endSearch();

10. var chooseBin = new IloOplRunConfiguration("select.mod");
11. chooseBin.cplex = cplex;
12. chooseBin.oplModel.addDataSource(data);
13. chooseBin.oplModel.generate();
14. chooseBin.cplex.solve();
15. chooseBin.oplModel.postProcess();
```

Fig. 4 ILOG Script for OPL example

the actual generation model, while lines 02–04 set up and start the search. The loop of line 05 asks for all solutions and adds them (line 07) to the `data` structure (`n` is the variable that tells how many objects of each material are in the bin). When no more solutions exist, line 09 ends the search. With all variables now available, line 10 associates the internal name `chooseBin` with the selection model and line 12 passes the data (variables) to it. Lines 14 and 15 solve the problem and wrap up.

This example generated all variables a priori before solving the optimization model. It is possible, of course, to implement an iterative hybrid algorithm that starts with a subset of the variables and transfers dual information (shadow prices) to the generation problem so that it can look for a new (improving) variable to be added to the optimization (master) problem. Then, as usual, this process is repeated until no improving variables exist.

Table 5 ILOG CP Optimizer/OPL Data Sheet

Feature	Description
Tool name	ILOG CP Optimizer and OPL Development Studio
User interface	1a, 1b, 1c
Solver support	2a, 2c
Relaxation support	3a, 3c, 3d
Inference mechanisms	4a, 4b, 4c
Search mechanisms	5a, 5b
Decomposition mechanisms	6b, 6c
Search control	7a, 7b, 7c
Written in	C++
Supported platforms	AIX, Linux, Mac OS X, Solaris, Windows (full OPL IDE available only under Windows)
Availability	Commercial, academic and student licenses available. See http://www.ilog.com.
References	[58, 37, 36]
Notes	ILOG CP Optimizer only handles discrete decision variables

8 SCIP

SCIP (Solving Constraint Integer Programs [2, 4]) is an implementation of the Constraint Integer Programming (CIP) paradigm, which is an integration of CP, MILP, and SAT methodologies. It can be used either as a black box solver or as a framework by adding user plug-ins written in C or C++. SCIP allows total control of the solution process and access to detailed information from the solver. A number of predefined macros facilitate implementation by encapsulating commonly used function calls into a simpler interface. SCIP comes with more than 80 default plug-ins that turn it into a full solver for MILP and pseudo-Boolean optimization.

A Constraint Integer Program (CIP) is defined as the optimization problem

$$c^* = min\{cx \mid \mathscr{C}(x),\, x \in \mathbb{Z}^I \times \mathbb{R}^{N\setminus I}\}\,, \tag{4}$$

where $\mathscr{C} = \{C_1, \ldots, C_m\}$ is a set of constraints, and $I \subseteq N = \{1, \ldots, n\}$. By definition, the constraint set $\mathscr{C}$ has to be such that, once the integer variables have been assigned values, the remaining problem becomes a linear program.

SCIP is a framework for branching, cutting, pricing and propagation, and its implementation is based on the idea of plug-ins, which makes it very flexible and extensible. Here is a list of the main types of SCIP plug-ins and their roles:

Constraint handlers: These are the central objects of SCIP. Each constraint handler represents the semantics of a single class of constraints and provides algorithms to handle constraints of the corresponding type. The primary task of a constraint handler is to check a given solution for feasibility with respect to all constraints of its type existing in the problem instance. To improve the performance of the solving process, constraint handlers may provide additional algorithms and information about their constraints to the framework, such as: pre-solving methods to simplify the problem, propagation methods to tighten variable domains, linear relaxations, branching decisions and separation routines

Domain propagators: Constraint based domain propagation is supported by the constraint handler concept of SCIP. In addition, SCIP features two dual domain reduction methods that are driven by the objective function, namely objective propagation and root reduced-cost strengthening.

Conflict analyzers: SCIP generalizes conflict analysis to CIP and, as a special case, to MIP [1]. There are two main differences between CIP and SAT solving in the context of conflict analysis: CIP variables are not necessarily binary and the infeasibility of a subproblem in the CIP search tree is usually caused by the LP relaxation of that subproblem. Because it is NP-hard to identify a subset of the local bounds of minimal cardinality that make the LP infeasible, SCIP uses a greedy heuristic approach based on an unbounded ray of the dual LP.

Cutting plane separators: SCIP features separators for a myriad of cuts ([10, 29, 34, 39, 41, 49, 52]) in addition to $\{0, 1/2\}$-cuts and multi-commodity-flow cuts. For a survey, see [64]. For cut selection, SCIP uses *efficacy* and *orthogonality* (see [11, 6]), and parallelism with respect to the objective function.

Primal heuristics: SCIP has 23 different heuristics, which can be classified into four categories: rounding, diving, objective diving, and improvement ([15]).

Node selectors and branching rules: SCIP implements most of the well-known branching rules, including reliability branching ([5]) and hybrid branching ([3]), and it allows the user to implement arbitrary branching schemes. Several node selection strategies are pre-defined, such as depth-first, best-first, and best-estimate ([27]). The default search strategy is a combination of these three.

Pre-solving: SCIP implements a full set of primal and dual pre-solving reductions for MIP problems, such as removing redundant constraints, fixing variables, strengthening the LP relaxation by exploiting integrality information, improving constraint coefficients, clique extractions, etc. It also uses the concept of *restarts*, which are a well-known ingredient of modern SAT solvers.

Additional SCIP features include:

- Variable pricers to dynamically create problem variables
- Relaxators to provide relaxations and dual bounds in addition to the LP relaxation, e.g. semi-definite or Lagrangian, working in parallel or interleaved
- Dynamic cut pool management
- Counting of feasible solutions, visualization of the search tree, and customization of output statistics

The fundamental search strategy in SCIP is the exploration of a branch-and-bound tree. All involved algorithms operate on a single search tree, which allows for a very close interaction between different constraint handlers. SCIP manages the branching tree along with all subproblem data, automatically updates the LP relaxation, and handles all necessary transformations due to problem modifications during the presolving stage. Additionally, a cut pool, cut filtering, and a SAT-like conflict analysis mechanism are available. SCIP provides its own memory management and plenty of statistical output.

In addition to allowing the user to integrate MILP, CP and SAT techniques, SCIP can also be used as a pure MILP or as a pure CP solver. Computational results shown on the benchmark web pages of Hans Mittelmann[1] indicate that SCIP is one of the fastest non-commercial MILP solvers currently available.

Table 6 SCIP Data Sheet

Feature	Description
Tool name	SCIP – Solving Constraint Integer Programs
User interface	1c
Solver support	2a, 2c, 2d
Relaxation support	3a, 3c, 3d
Inference mechanisms	4a, 4b, 4c, 4d
Search mechanisms	5a, 5b, 5c
Decomposition mechanisms	6b
Search control	7c
Written in	C
Supported platforms	Should compile with any ANSI C compiler
Developers	Tobias Achterberg, Timo Berthold, Stefan Heinz, Thorsten Koch and Kati Wolter
Availability	SCIP is distributed under the ZIB Academic License (see http://zibopt.zib.de/academic.txt). Source code and binaries are available at http://scip.zib.de.
References	[2, 4]
Notes	SCIP is part of the ZIB Optimization Suite. Also available on the NEOS Server.

SCIP uses an external LP solver to handle the LP relaxations. The current release (1.1.0) can interface with CPLEX, Xpress-MP, Mosek, SoPlex, and the COIN-OR

[1] http://plato.asu.edu/ftp/milpf.html

LP solver CLP. The ZIB Optimization Suite (http://zibopt.zib.de) offers a complete integrated bundle of SCIP, SoPlex and ZIMPL (a MILP modeling language). Furthermore, pre-compiled binaries including SoPlex or CLP can be found on the SCIP web page.

Coming soon to SCIP: interface to the LP solver QSopt, support for FlatZinc [47] models; improved pseudo-boolean performance. Further down the road: exact integer programming techniques (sound solver without rounding errors); additional global CP constraints; MINLP capabilities (non-linear and non-convex constraints).

9 SIMPL

SIMPL ([8, 65]) is based on two principles: algorithmic unification and constraint-based control. Algorithmic unification begins with the premise that integration should occur at a fundamental and conceptual level, rather than postponed to the software design stage. Optimization methods and their hybrids are viewed, to the extent possible, as special cases of a single solution method that can be adjusted to exploit the structure of a given problem. This goal is addressed with a *search-infer-and-relax* algorithmic framework, coupled with *constraint-based control* in the modeling language. The search-infer-and-relax scheme encompasses a wide variety of methods, including branch-and-cut (B&C) methods for integer programming, branch-and-infer methods for constraint programming, popular methods for continuous global optimization, nogood-based methods as Benders decomposition and dynamic backtracking, and even heuristic methods such as local search and greedy randomized adaptive search procedures (GRASPs) [26].

Constraint-based control allows the design of the model itself to tell the solver how to combine techniques so as to exploit problem structure. Highly-structured subsets of constraints are written as metaconstraints, which are similar to global constraints in constraint programming. Syntactically, a metaconstraint is written much as linear or global constraints are written, but it is accompanied by parameters that specify how the constraint is to be implemented during the solution process. A metaconstraint may specify how it is to be relaxed, how it will filter domains, and/or how the search procedure will branch when the constraint is violated. When such parameters are omitted, a pre-specified default behavior is used.

The relaxation, inference, and branching techniques are devised for each constraint's particular structure. For example, a metaconstraint may be associated with a tight polyhedral relaxation from the integer programming literature and/or an effective domain filter from constraint programming. Because constraints also control the search, if a branching method is explicitly indicated for a metaconstraint, the search will branch accordingly. The selection of metaconstraints to formulate the problem determines how the solver combines algorithmic ideas to solve the problem.

To illustrate the above ideas, we consider the following integer knapsack problem with a side constraint (see Chap. 2 of [32]):

$$\begin{aligned} \min\ & 5x_1 + 8x_2 + 4x_3 \\ \text{s.t. } & 3x_1 + 5x_2 + 2x_3 \geq 30 \\ & \textit{alldifferent}(x_1, x_2, x_3) \\ & x_j \in \{1,2,3,4\},\ \forall\ j \end{aligned}$$

A SIMPL model for the above problem is shown in Fig. 5. The model starts with a `DECLARATIONS` section in which constants and variables are defined. Line 07 defines the objective function. In the `CONSTRAINTS` section, the two constraints of the problem are represented by the (named) metaconstraints `totweight` and `distinct`, and their definitions show up in lines 10 and 13, respectively. The `relaxation` statements in lines 11 and 14 indicate the relaxations to which those constraints should be posted. Both constraints will be present in the LP and in the CP relaxations. Because the `alldifferent` constraint is not linear, submitting it to an LP relaxation means that it will be automatically transformed into a linear approximation (in this case, the convex hull formulation) of the set of its feasible solutions (see [63]). In the `SEARCH` section, line 16 indicates we will do branch-and-bound (`bb`) with depth-first search (`depth`). The `branching` statement in line 17 says that we will branch on the first of the `x` variables that is not integer (branching on a variable means branching on its indomain constraint). Once all `x`'s are integer, the most violated of the `alldifferent` constraints will be used for branching (`distinct:most`). Initially, bounds consistency maintenance in the

```
01. DECLARATIONS
02.    n = 3; limit = 30;
03.    cost[1..n] = [5,8,4]; weight[1..n] = [3,5,2];
04.    discrete range xRange = 1 to 4;
05.    x[1..n] in xRange;
06. OBJECTIVE
07.    min sum i of cost[i]*x[i]
08. CONSTRAINTS
09.    totweight means {
10.       sum i of weight[i]*x[i] >= limit
11.       relaxation = {lp, cp} }
12.    distinct means {
13.       alldifferent(x)
14.       relaxation = {lp, cp} }
15. SEARCH
16.    type = {bb:depth}
17.    branching = {x:first, distinct:most}
```

Fig. 5 SIMPL model for the hybrid knapsack problem

CP solver removes value 1 from the domain of x_2 and the solution of the LP relaxation is $x = (2\frac{2}{3}, 4, 1)$. After branching on $x_1 \leq 2$, bounds consistency determines that $x_1 \geq 2$, $x_2 \geq 4$ and $x_3 \geq 2$. At this point, the `alldifferent` constraint pro-

duces further domain reduction, yielding the feasible solution $(2,4,3)$. Notice that no LP relaxation had to be solved at this node. In a similar fashion, the CP solver may be able to detect infeasibility even before the linear relaxation has to be solved.

In [65], SIMPL was used to model and solve four classes of problems that had been successfully solved by custom implementations of integrated approaches, namely: production planning, product configuration, machine scheduling, and truss structure design. Computational results indicate that the high-level models implemented in SIMPL either match or surpass the performance of the original special-purpose codes at a fraction of the implementation effort.

Table 7 SIMPL Data Sheet

Feature	Description
Tool name	SIMPL: A Modeling Language for Integrated Problem Solving
User interface	1b, 1c
Solver support	2a, 2c
Relaxation support	3a, 3d
Inference mechanisms	4a, 4b, 4c
Search mechanisms	5a
Decomposition mechanisms	6c
Search control	7a, 7c
Written in	C++
Supported platforms	Linux
Developers	Ionuţ Aron, John Hooker and Tallys Yunes
Availability	Free for academic use. Preliminary demo version available at http://moya.bus.miami.edu/∼tallys/simpl.php.
References	[8, 65]
Notes	Looking to expand the development group.

SIMPL requires external solvers to handle the problem relaxations. The current release (0.08.22) can use CPLEX as an LP solver and ECL$^{\text{i}}$PS$^{\text{e}}$ as a CP solver. Other solvers can be added by deriving an object class and implementing standard interface methods.

Coming soon to SIMPL: Interfaces to NLP solvers; additional LP and CP solvers; pre-processing; cutting plane generation. Further down the road: support for SAT solvers; local and incomplete search; branch-and-price; high-level search language; GUI and integrated development environment.

10 Xpress-Mosel

The Xpress-Mosel language [22] (Mosel for short) is a modeling language that is part of the FICO Xpress suite of mathematical modeling and optimization tools[2]. It

[2] Originally developed by Dash Optimization under the name Xpress-MP.

allows the user to formulate the problem, solve it with a suitable solver engine, and analyze the solution, using a fully-functional programming language specifically designed for this purpose. Mosel programs can be run interactively or embedded within an application. The language is integrated within the Xpress-IVE visual development environment. In addition to the usual features available in standard modeling languages, it provides support for arbitrary ranges, index sets, sparse objects, and a debugger that supports tracing and analyzing the execution of a model.

Mosel is an open, user-extensible language. The Mosel distribution includes extension libraries (so-called *modules*), one of which provides control of the Xpress-Optimizer (module *mmxprs*), through optimization statements in the Mosel program. Other solver modules give access to formulating and solving non-linear problems (module *mmnl* handles QCQP, MIQCQP and convex NLP), the Stochastic Programming tool Xpress-SP[3], and the Constraint Programming software Xpress-Kalis (module *kalis*). In Mosel, CP and MIP solving may be used *sequentially*, for instance, employing CP constraint propagation as a pre-processing routine for LP/MIP problems; or *in parallel* as would happen, for example, when CP solving is used as cut or variable generation routine during a MIP branch-and-bound search. We now provide a simplified example of the former type of integration to illustrate Mosel's modeling constructs (taken from [30, 31]). Due to space limitations we omit the data and variable declarations in our Mosel models.

Consider a project scheduling problem in which a set of tasks (`TASKS`) with a certain default duration (`DUR(i)`, in weeks) have to be executed (e.g. a real estate development project). Precedence constraints exist between given pairs of tasks (`ARC(i,j)`) and, if the manager is willing to spend extra money (`COST(i)`), it is possible to reduce the duration of each task by `save(i)`, up to a certain amount (`MAXW(i)`) (this is sometimes referred to as *crashing*). Assume that we know the earliest project completion date without crashing (`Finish`) and the client is willing to pay `BONUS` dollars for every week the work finishes early. To maximize the manager's profit we will solve this problem in two stages: first (Fig. 6), we use the CP solver to find bounds on the start times of each task (`start(i)`), and then we use those bounds in an LP optimization model (Fig. 7). In Fig. 6, line 02 indicates we will use the CP module and lines 04 and 05 declare the start time and duration variables. Note that the actual duration of a task is a variable because we do not know a priori by how many weeks each task will be crashed. The calculated bounds on task start times, which are declared in line 06, will be retrieved from the CP solver in lines 12–15, and passed to the LP model through a shared memory space in lines 16–18. Lines 08 and 09 declare the variable domains. Lines 10 and 11 create and post the precedence constraints. In Fig. 7, the master model includes the optimizer module in line 02 and declares a CP model in line 04. In lines 06 and 07, the CP model from Fig. 6 is compiled, loaded and run. Line 09 retrieves from the shared memory space the bounds on task start times obtained by the CP model. We declare the variables of the optimization model in lines 11–14, and the objective function in line 15. Task number `N` is a virtual task introduced so that its start time represents

[3] Recently turned into open-source and available from the Xpress website.

```
01. model "Crashing CP"
02. uses "kalis"
03. declarations
04.   start: array(TASKS) of cpvar
05.   duration: array(TASKS) of cpvar
06.   lbstart, ubstart: array(TASKS) of integer
07. end-declarations
08. forall(j in TASKS) setdomain(start(j),0,Finish)
09. forall(j in TASKS) setdomain(duration(j),
                                 DUR(j)-MAXW(j),DUR(j))
10. forall(i,j in TASKS | exists(ARC(i,j)))
11.   start(i) + duration(i) <= start(j)
12. forall(i in TASKS) do
13.   lbstart(i) := getlb(start(i))
14.   ubstart(i) := getub(start(i))
15. end-do
16. initializations to "raw:"
17.   lbstart as "shmem:lbstart" ubstart as "shmem:ubstart"
18. end-initializations
19. end-model
```

Fig. 6 Mosel CP model for project crashing. File name: crash1.mos

```
01. model "Crashing master (CP + LP)"
02. uses "mmxprs", "mmjobs"
03. declarations
04.   CPmodel: Model
05. end-declarations
06. res := compile("crash1.mos"); load(CPmodel,"crash1.bim")
07. run(CPmodel); wait
08. initializations from "raw:"
09.   lbstart as "shmem:lbstart" ubstart as "shmem:ubstart"
10. end-initializations
11. declarations
12.   start: array(TASKS) of mpvar
13.   save: array(TASKS) of mpvar
14. end-declarations
15. Profit := BONUS*save(N) - sum(i in 1..N-1) COST(i)*save(i)
16. forall(i,j in TASKS | exists(ARC(i,j)))
17.   Precm(i,j) := start(i) + DUR(i) - save(i) <= start(j)
18. start(N) + save(N) = Finish
19. forall(i in 1..N-1) save(i) <= MAXW(i)
20. forall(i in 1..N-1) do
21.   lbstart(i) <= start(i); start(i) <= ubstart(i)
22. end-do
23. maximize(Profit)
24. end-model
```

Fig. 7 Mosel LP model for project crashing

the completion time of the project. Lines 17 and 18 state the precedence constraints and tie the project completion time to the known duration `Finish`. Line 19 limits the crash amounts, and lines 20–22 use the bounds retrieved from the CP model to tighten the domain of `start(i)`.

Two successful, and more intricate, implementations of hybrid algorithms using Mosel appeared in [17] and [51].

Table 8 Xpress-Mosel Data Sheet

Feature	Description
Tool name	Xpress-Mosel
User interface	1a, 1b, 1c
Solver support	2a, 2b, 2c
Relaxation support	3a, 3b, 3d
Inference mechanisms	4a, 4b, 4c
Search mechanisms	5a, 5b
Decomposition mechanisms	6a, 6b, 6c
Search control	7a, 7b, 7c
Supported platforms	AIX, HP-UX, Linux, Solaris, Windows (Xpress-IVE available on Windows only)
Main developer	Yves Colombani
Availability	Commercial, academic and student licenses available. See http://www.dashoptimization.com.
References	[22, 30, 21]

Coming soon to Xpress-Mosel: support for multiple problems in the same model file, introducing the concept of *local scoping* i.e. variables and constraints are no longer necessarily global and instead can be defined in the context of a specific sub-problem (useful in the implementation of user-defined heuristics); a new feature of Xpress-Kalis that introduces the possibility to work with automatic LP/MIP relaxations of CP constraints (linear constraints and global constraints like alldifferent, occurrence, etc.). the LP/MIP representation is generated automatically and solved by Xpress-Optimizer with numerous configuration options as to how, where and when to solve the MP problem(s).

11 Other Hybrid Tools

This section covers a few other software tools that include some level of support for integration.

11.1 COIN-OR

COIN-OR (Computational Infrastructure for Operations Research) [40] is a project aimed at spurring the development of open-source software for the OR community. Its objectives include speeding the development and deployment of models, algorithms, and cutting-edge computational research, as well as providing a forum for peer review of software similar to that provided by archival journals for theoretical research. While not being a hybrid tool per se, the COIN-OR initiative can be viewed as a hybrid set of tools that could, in principle, be combined. The COIN-OR repository is composed of many different projects divided into categories such as developer tools, graph algorithms, abstract interfaces (e.g. OSI), metaheuristics, and optimization packages that can handle distinct types of problems (deterministic linear and non-linear (both continuous and discrete), deterministic semi-definite continuous, and stochastic). For further details, see http://www.coin-or.org.

11.2 Microsoft Solver Foundation

Recently, the Microsoft Solver Foundation (version 1.1) has added links to a number of well known solvers through its Solver Plug-in System, namely CPLEX, Xpress-MP, Mosek and Gurobi. There is also support for CP solvers. For more details see http://www.solverfoundation.com.

11.3 Prolog IV

Prolog IV [46] is an ISO-compliant replacement for the Prolog III language. It incorporates all the main features of Prolog III with some important changes. It allows programmers to express a wide variety of constraints over real and rational numbers, integers (finite domains), booleans and lists. In addition to expressing classical linear programming problems on discrete and continuous quantities, it permits, among other things, the use of mixed real/integers problems and the use of boolean operations to formalize constraint disjunctions. The algorithms include a non-optimized algorithm for lists (different from Prolog III), Gauss and Simplex algorithms for equations and linear inequalities over rationals, and an interval method for approximate solving of non-linear constraints over reals. The compiler is integrated into a complete graphic programming environment featuring tools such as a project editor, a multi-window text editor, grapher, debugger, and on-line help.

11.4 SALSA

SALSA [38] is a language dedicated to specifying local, global and hybrid search algorithms. It provides the user with the ability to specify the way the global search tree is explored. SALSA attempts to make the creation of hybrid algorithms a less tedious and less error-prone task by providing a high-level language that offers the ability to perform non-monotonic operations and hypothetical reasoning. It proposes to consider logic and control separately and, because it is not a standalone language, it works in cooperation with a host programming language. SALSA allows the programmer to specify the choice mechanisms that are responsible for generating the moves of a search algorithm in an elegant manner, and it offers primitives for composing the transitions from one state to the next. In global search, goals and constraints describe properties of final states, while in local search, invariants and neighborhoods describe properties of all states. However, because the basic branching mechanism of global search and local search algorithms are very much alike, a language expressing neighborhoods and choices in the same formalism would be able to express hybrid combinations. This is what SALSA does.

11.5 ToOLS

The purpose of ToOLS [23] (Templates of On-Line Search) is twofold: to help a constraint programmer to build complex customized search algorithms, and to offer ready-made search components for engineers, improving algorithm reuse and capitalization. ToOLS is part of a finite-domain constraint solver library called Eclair, developed in the high-level language Claire [18]. ToOLS divides the description of a search algorithm into three parts (or components): a complete search tree defined by a refinement-based search scheme, a set of conditions restricting the exploration of the tree, and a combination of several partial explorations. Each component can also be reused separately. A search algorithm is a Claire object created by a functional composition of constructors called ToOLS *primitives*. A special function (solve, solveAll or minimize) specifies the goal of the search (satisfaction or optimization) and is applied to a single algorithm object. ToOLS has been successfully used to implement large neighborhood search methods applied to satellite observation scheduling and military applications.

12 Conclusion

The availability of software tools for integrated optimization has dramatically increased since the first CP-AI-OR workshop held in 1999. Today, it is possible to exploit the power of hybrid algorithms without having to spend several days (or even months) writing and debugging computer code. Both academia and industry

recognize the need for better optimization software that facilitates the integration of solution techniques, and this chapter provides a brief overview of some existing software packages.

Despite the tremendous progress over the past ten years, there is still much room for improvement. In [60], Wallace, Caseau and Puget mention that "to make an impact, our technology must be made useable so that (i) highly qualified experts can develop solutions very quickly", and "(ii) less expert users can also exploit the technology successfully." This greater *accessibility* was one of the fundamental forces behind the increased adoption and dissemination of traditional OR techniques. Moreover, the authors also make a case for the need to "make the user's task more manageable". We could, for instance, "identify general problem features that correlate with suitable algorithms. Define simple rules about when to use one problem solving method and when to use another one. Categorise which forms of hybridisation work best for which kinds of problems and algorithms". This means that our software tools could greatly benefit from a database of *meta models* (see [19]). Over the years, the OR and CP communities have gathered a wealth of knowledge about what works and what does not work; what kinds of mathematical formulas are more suitable representations of certain real-life phenomena; what kinds of symmetries typically arise in a given class of problems or from a given choice of variables; which cutting planes are effective for a given class of optimization problem; etc. All of this knowledge resides in the minds of our best modelers when it should be residing in the software itself. The next generation of modeling tools needs to extract more information from the user. The creation of a new model should start with an electronic questionnaire that will collect information about the nature and structure of the problem. It could, for instance, present a tree hierarchy to the user and allow him/her to click on the nodes that have a relationship to that particular problem, as depicted in Fig. 8. To input a problem that combines location,

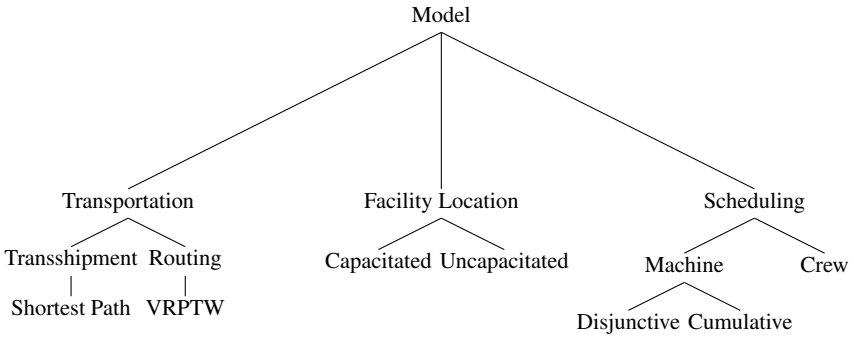

Fig. 8 Hypothetical hierarchy of models. Users click on the nodes that are relevant to their problem.

routing and scheduling aspects, the user simply clicks on more than one node. Similar hierarchies can be used to gather information about the problem domain (e.g.

manufacturing, finance, marketing), as well as the nature of the objective function and constraints (e.g. does it include uncertainty/randomness?). In possession of this extra knowledge, the modeling tool will be in a better position to choose its own default parameters, to suggest modeling constructs (constraints, cuts, relaxations, decompositions), and even to flag potentially ineffective choices made by the users as they input the model.

In [33], Hooker argues that "we should take full advantage of the graphical user interface to empower modelers. A metaconstraint should be invoked by opening a window on the computer screen, not by typing a statement. The window should present various options for refining the constraint and importing data. The model as a whole should be depicted graphically, with an opportunity to click on modules for a more detailed look." Once again, it all boils down to accessibility. By making our models easier to build, and therefore easier to understand, we can reach out to a larger audience. Software packages that implement some of these ideas include Visual CHIP [9], CHIP Factory [53], and the computer simulation software Arena [35]. Visualization also plays an important role in solution analysis and model tuning. The visualization capabilities of tools like CHIP and Comet represent an important step in that direction. When combined with the kind of meta-modeling information discussed above, such a graphical modeling environment would be able to pre-populate its window with some of the necessary modules. All of this automatic behavior, of course, must be available while also staying out of the way of the expert user.

The development and growth of hybrid modeling tools has created exciting new challenges and numerous research directions. The general trend seems to be one of *unification* rather than separation. Over the next ten years, our tools will be able to do more, while asking for less of our guidance. It is unlikely that the software expertise will ever substitute the human expertise, but significant improvements over the current state-of-the-art are definitely possible. By being aware of these strengths and weaknesses, we have taken an important step toward a new generation of software tools for integrated optimization.

Acknowledgments

The author would like to thank Timo Berthold, Stefan Heinz, Susanne Heipcke, Katya Krasilnikova, Michela Milano, Philippe Refalo, Nick Sahinidis, Kish Shen, Helmut Simonis, Peter Stuckey, Pascal Van Hentenryck, and Mark Wallace for answering technical questions, and for providing feedback on the presentation of the material and on the accuracy of the information about each software tool.

References

1. Achterberg, T.: Conflict analysis in mixed integer programming. Discrete Optimization **4**(1), 4–20 (2007). Special issue: Mixed Integer Programming
2. Achterberg, T.: SCIP: Solving constraint integer programs. Mathematical Programming Computation **1** (2008)
3. Achterberg, T., Berthold, T.: Hybrid branching. In: W.J. van Hoeve, J. Hooker (eds.) Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 6th International Conference, CPAIOR 2009, *Lecture Notes in Computer Science*, vol. 5547, pp. 309–311. Springer (2009)
4. Achterberg, T., Berthold, T., Koch, T., Wolter, K.: Constraint integer programming: a new approach to integrate CP and MIP. In: L. Perron, M. Trick (eds.) Proceedings of the Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR), *Lecture Notes in Computer Science*, vol. 5015, pp. 6–20. Springer-Verlag (2008)
5. Achterberg, T., Koch, T., Martin, A.: Branching rules revisited. Operations Research Letters **33**, 42–54 (2005)
6. Andreello, G., Caprara, A., Fischetti, M.: Embedding cuts in a branch and cut framework: a computational study with $\{0, 1/2\}$-cuts. INFORMS Journal on Computing **19**(2), 229–238 (2007)
7. Apt, K.R., Wallace, M.: Constraint Logic Programming Using ECL$^{\text{i}}$PS$^{\text{e}}$. Cambridge University Press (2007)
8. Aron, I.D., Hooker, J.N., Yunes, T.H.: SIMPL: A system for integrating optimization techniques. In: J. Rgin, M. Rueher (eds.) Proceedings of the Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR), *Lecture Notes in Computer Science*, vol. 3011, pp. 21–36. Springer-Verlag (2004)
9. Baader, F., Comon, H., Smolka, G.: Visual CHIP: A visual language for defining constraint programs. In: Annual Workshop of the ESPRIT Working Group "Constructions of Computational Logic II" (CCL). Dagstuhl (1997)
10. Balas, E.: Facets of the knapsack polytope. Mathematical Programming **8**, 146–164 (1975)
11. Balas, E., Ceria, S., Cornuéjols, G.: Mixed 0-1 programming by lift-and-project in a branch-and-cut framework. Management Science **42**, 1229–1246 (1996)
12. Barnhart, C., Johnson, E.L., Nemhauser, G.L., Savelsbergh, M.W.P., Vance, P.H.: Branch-and-price: Column generation for solving huge integer programs. Operations Research **46**, 316–329 (1998)
13. Becket, R., Brand, S., Brown, M., Duck, G.J., Feydy, T., Fischer, J., Huang, J., Marriott, K., Nethercote, N., Puchinger, J., Rafeh, R., Stuckey, P.J., Wallace, M.G.: The many roads leading to Rome: Solving Zinc models by various solvers. In: Proceedings of the 7th International Workshop on Constraint Modeling and Reformulation (ModRef) (2008)
14. Benders, J.F.: Partitioning procedures for solving mixed-variables programming problems. Numerische Mathematik **4**, 238–252 (1962)
15. Berthold, T.: Primal heuristics for mixed integer programs. Master's thesis, Technische Universität Berlin (2006)
16. Bocca, J.: Megalog — a platform for developing knowledge base management systems. In: Proceedings of the Second International Symposium on Database Systems for Advanced Applications (DASFAA). Tokyo, Japan (1991)
17. Bockmayr, A., Pisaruk, N.: Detecting infeasibility and generating cuts for MIP using CP. In: Fifth International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR), pp. 24–34. Montréal, Canada (2003)
18. Caseau, Y., Josset, F.X., Laburthe, F.: CLAIRE: combining sets, search and rules to better express algorithms. In: Proceedings of the International Conference on Logic programming (ICLP), pp. 245–259 (1999)

19. Caseau, Y., Silverstein, G., Laburthe, F.: Learning hybrid algorthms for vehicle routing problems. Theory and Practice of Logic Programming **1**(6), 779–806 (2001)
20. Clocksin, W.F., Mellish, C.S.: Programming in Prolog. Springer Verlag (1981)
21. Colombani, Y., Daniel, B., Heipcke, S.: Mosel: a Modular Environment for Modeling and Solving Problems. In: J. Kallrath (ed.) Modeling Languages in Mathematical Optimization, pp. 211–238. Kluwer Academic Publishers, Boston (2004)
22. Colombani, Y., Heipcke, S.: Mosel: An extensible environment for modeling and programming solutions. In: Proceedings of the International Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR) (2002)
23. de Givry, S., Jeannin, L.: ToOLS: a library for partial and hybrid search methods. In: Fifth International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR), pp. 124–138. Montréal, Canada (2003)
24. Dincbas, M., Van Hentenryck, P., Simonis, M., Aggoun, A., Graf, T., Berthier, F.: The constraint logic programming language CHIP. In: Proceedings of the International Conference of Fifth Generation Computer Systems, pp. 693–702 (1988)
25. Duck, G.J., Stuckey, P.J., Brand, S.: ACD term rewriting. In: S. Etalle, M. Truszczynski (eds.) Proceedings of the 22nd International Conference on Logic Programming (ICLP), *Lecture Notes in Computer Science*, vol. 4079, pp. 117–131. Springer-Verlag (2006)
26. Feo, T., Resende, M.: Greedy randomized adaptive search procedures. Journal of Global Optimization **6**, 109–133 (1995)
27. Forrest, J.J., Hirst, J.P.H., Tomlin, J.A.: Practical solution of large scale mixed integer programming problems with UMPIRE. Management Science **20**(5), 736–773 (1974)
28. Garcia de la Banda, M., Marriott, K., Rafeh, R., Wallace, M.: The modelling language Zinc. In: F. Benhamou (ed.) Proceedings of the 12th International Conference on Principles and Practice of Constraint Programming (CP), *Lecture Notes in Computer Science*, vol. 4204, pp. 700–705. Springer-Verlag (2006)
29. Gomory, R.E.: Solving linear programming problems in integers. In: R. Bellman, J.M. Hall (eds.) Symposia in Applied Mathematics X, Combinatorial Analysis, pp. 211–215. American Mathematical Society (1960)
30. Guéret, C., Heipcke, S., Prins, C., Sevaux, M.: Applications of Optimization with Xpress-MP. Dash Optimization, Blisworth, UK (2002). URL http://www.dashoptimization.com/applications_book.html
31. Heipcke, S.: Hybrid MIP/CP solving with Xpress-Optimizer and Xpress-Kalis. Xpress Whitepaper, FICO (2005). URL http://www.dashoptimization.com
32. Hooker, J.N.: Logic-Based Methods for Optimization: Combining Optimization and Constraint Satisfaction. Wiley-Interscience Series in Discrete Mathematics and Optimization (2000)
33. Hooker, J.N.: Good and bad futures for constraint programming (and operations research). Constraint Programming Letters **1**, 21–32 (2007). Special Issue on the Next 10 Years of Constraint Programming
34. Johnson, E.L., Padberg, M.W.: Degree-two inequalities, clique facets and biperfect graphs. Annals of Discrete Mathematics **16**, 169–187 (1982)
35. Kelton, W.D., Sadowski, R.P., Sturrock, D.T.: Simulation with Arena, 4th edn. McGraw Hill (2007)
36. Laborie, P.: IBM ILOG CP Optimizer for detailed scheduling illustrated on three problems. In: J.N. Hooker, W.J. van Hoeve (eds.) Proceedings of the Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR), Lecture Notes in Computer Science. Springer-Verlag (2009). To appear.
37. Laborie, P., Rogerie, J., Shaw, P., Vilím, P., Wagner, F.: ILOG CP Optimizer: detailed scheduling model and OPL formulation. Tech. Rep. 08-002, ILOG (2008). Available at http://www2.ilog.com/techreports/
38. Laburthe, F., Caseau, Y.: SALSA: a language for search algorithms. Constraints **7**(3-4), 255–288 (2002)

39. Lechford, A.N., Lodi, A.: Strengthening Chvátal-Gomory cuts and Gomory fractional cuts. Operations Research Letters **30**(2), 74–82 (2002)
40. Lougee-Heimer, R.: The Common Optimization INterface for Operations Research. IBM Journal of Research and Development **47**(1), 57–66 (2003)
41. Marchand, H.: A polyhedral study of the mixed knapsack set and its use to solve mixed integer programs. Ph.D. thesis, Faculté des Sciences Appliquées, Université Catholique de Louvain (1998)
42. McCormick, G.P.: Computability of global solutions to factorable nonconvex programs: Part I - convex underestimating problems. Mathematical Programming **10**, 147–175 (1976)
43. McCormick, G.P.: Nonlinear Programming: Theory, Algorithms and Applications. Wiley-Interscience, New York (1983)
44. Meier, M., Kay, P., Van Rossum, E., Grant, H.: SEPIA programming environment. In: Proceedings of the Workshop on PROLOG Programming Environments NACLP'89, pp. 82–86 (1989)
45. Minton, S., Johnson, M.D., Philips, A.B.: Solving large-scale constraint satisfaction and scheduling problems using a heuristic repair method. In: Proceedings of the Eigth National Conference on Artificial Intelligence (AAAI-90), pp. 17–24. Boston (1990)
46. Narboni, G.: From prolog III to prolog IV: The logic of constraint programming revisited. Constraints **4**(4), 313–335 (1999)
47. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: Towards a standard CP modelling language. In: C. Bessière (ed.) Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP), *Lecture Notes in Computer Science*, vol. 4741, pp. 529–543. Springer-Verlag (2007)
48. Ohrimenko, O., Stuckey, P.J., Codish, M.: Propagation = lazy clause generation. In: C. Bessière (ed.) Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP), *Lecture Notes in Computer Science*, vol. 4741, pp. 544–558. Springer-Verlag (2007)
49. Padberg, M.W., van Roy, T.J., Wolsey, L.A.: Valid inequalities for fixed charge problems. Operations Research **33**(4), 842–861 (1985)
50. Puchinger, J., Stuckey, P.J., Wallace, M., Brand, S.: From high-level model to branch-and-price solution in G12. In: L. Perron, M. Trick (eds.) Proceedings of the Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR), *Lecture Notes in Computer Science*, vol. 5015, pp. 218–232. Springer-Verlag (2008)
51. Sadykov, R.: A hybrid branch-and-cut algorithm for the one-machine scheduling problem. In: J. Régin, M. Rueher (eds.) Proceedings of the Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR), *Lecture Notes in Computer Science*, vol. 3011, pp. 409–414. Springer-Verlag (2004)
52. Savelsbergh, M.W.P.: Preprocessing and probing techniques for mixed integer programming problems. ORSA Journal on Computing **6**, 445–454 (1994)
53. Simonis, H.: Finite domain constraint programming methodology. In: Second International Conference and Exhibition on the Practical Application of Constraint Technologies and Logic Programming (PACLP). Manchester, U.K. (2000). Tutorial
54. Somogyi, Z., Henderson, F., Conway, T.: The execution algorithm of Mercury, an efficient purely declarative logic programming language. Journal of Logic Programming **29**(1-3), 17–64 (1996)
55. Stuckey, P.J., Garcia de la Banda, M., Maher, M.J., Marriott, K., Slaney, J.K., Somogyi, Z., Wallace, M., Walsh, T.: The G12 project: Mapping solver independent models to efficient solutions. In: P. van Beek (ed.) Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming (CP), *Lecture Notes in Computer Science*, vol. 3709, pp. 13–16. Springer-Verlag (2005)
56. Tack, G.: Constraint propagation - models, techniques, implementation. Ph.D. thesis, Saarland University, Germany (2009). URL http://www.gecode.org
57. Tawarmalani, M., Sahinidis, N.V.: Global optimization of mixed-integer nonlinear programs: A theoretical and computational study. Mathematical Programming **99**, 563–591 (2004)

58. Van Hentenryck, P., Lustig, I., Michel, L., Puget, J.F.: The OPL Optimization Programming Language. MIT Press (1999)
59. Van Hentenryck, P., Michel, L.: Constraint-Based Local Search. MIT Press (2005)
60. Wallace, M., Caseau, Y., Puget, J.F.: Open perspectives. In: M. Milano (ed.) Constraint and Integer Programming: Toward a Unified Methodology, pp. 331–365. Kluwer (2004)
61. Wallace, M., Novello, S., Schimpf, J.: ECL$^{\text{i}}$PS$^{\text{e}}$: A platform for constraint logic programming. ICL Systems Journal **12**, 159–200 (1997)
62. Warren, D.H.D.: An abstract prolog instruction set. Tech. Rep. 309, SRI International (1983). http://www.ai.sri.com/pubs/files/641.pdf
63. Williams, H.P., Yan, H.: Representations of the all_different predicate of constraint satisfaction in integer programming. INFORMS Journal on Computing **13**(2), 96–103 (2001)
64. Wolter, K.: Implementation of cutting plane separators for mixed integer programs. Master's thesis, Technische Universität Berlin (2006)
65. Yunes, T., Aron, I.D., Hooker, J.N.: An integrated solver for optimization problems. Operations Research (2009). Forthcoming

Index

A

AIMMS 6
Arena 28
Assembly 8

B

BARON 4

C

C++ 2, 8, 21
Cadmium 11
CHIP 9, 28
  CHIP Factory 28
  Visual CHIP 28
Claire 26
COIN-OR 25
  CLP 9, 10, 13, 19
  OSI 13, 25
Comet 6, 28
CONOPT 6
constraint
  2-D alldifferent 10
  cardinality 10
  knapsack 20
  sequence 10
CPLEX 6, 10, 13, 18, 21, 25
cut
  multi-commodity flow 17

E

Eclair 26
ECL^{i}PS^{e} 9, 13, 21

F

FlatZinc 12, 19

G

G12 11
GAMS 6
Gecode 10, 13
GLPK 13
GRASP 19
Gurobi 25

I

IBM ILOG *see* ILOG
ILOG
  CP Optimizer 14
  OPL 14

J

Java 2

L

Lazy-FD 13
LP_SOLVE 9

M

Megalog 9
Mercury 11
MiniSAT 13
MiniZinc 11
MINOS 6
model
  annotation 10, 11, 13
Mosek 18, 25

N

NEOS 6, 18

O

OSL 6

P

Prolog 2, 9, 11
Prolog IV 25

Q

QSopt 19

S

SALSA 26
SCIP 16
SDPA 6
SEPIA 9
SIMPL 19
SNOPT 6
SoPlex 18

T

TiniSAT 13
ToOLS 26

X

Xpress-Mosel 21
Xpress-MP 10, 18, 25

Z

Zinc 11