Handbook of Constraint Programming

Francesca Rossi, Peter van Beek, Toby Walsh

Elsevier

Contents

Part I

First part

Chapter 1

Constraints in Procedural and Concurrent Languages

Thom Frühwirth, Laurent Michel, Christian Schulte

This chapter addresses the integration of constraints and search into programming languages from three different points of views. It first focuses on the use of constraints to model combinatorial optimization problem and to easily implement search procedures, then it considers the use of constraints for supporting concurrent computations and finally turns to the use of constraints to enable open implementations of constraints solvers.

The idea of approaching hard combinatorial optimization problems through a combination of search and constraint solving appeared first in logic programming. The genesis and growth of constraint programming within logic programming is not surprising as it catered to two fundamental needs: a declarative style and non-determinism.

Despite the continued support of logic programming for constraint programmers, research efforts were initiated to import constraint technologies into other paradigms (in particular procedural and object-oriented paradigms) to cater to a broader audience and leverage constraint-based techniques in novel areas. The first motivation behind a transition is a desire to ease the adoption of a successful technology. Moving constraints to a platform and paradigm widely accepted would facilitate their adoption within existing software systems by reducing resistance to the introduction of technologies and tools perceived as radically different. A second motivation is that constraints are versatile abstractions equally well suited for modeling and supporting concurrency. In particular, concurrent computation can be seen as agents that communicate and coordinate through a shared constraint store. Third, constraint-based techniques can leverage the semantics of a target application domain to design specialized constraints or search procedures that are more effective than off-the-shelves constraints. The ability, for domain specialists, to *easily* create, customize and extend both constraints solvers and search is therefore a necessity for adaptability.

The continued success and growth of constraints depends on the availability of flexible, extensible, versatile and easy to use solvers. It is contingent on retaining performance that rival or exceed the ad-hoc methods they supplant. Therefore, efficiency remains a key objective, often at odds with flexibility and ease of use.

Meeting these broad objectives, namely ubiquity, flexibility, versatility and efficiency within traditional paradigms that lack support for declarative programming creates unique challenges. First, a flexible tool must support mechanisms to let users define new constraints either through combinators or from first principles. The mechanisms should focus on the specification of *what* each constraint computes (its declarative semantics) rather than *how* it computes it (operational semantics) to retain simplicity without sacrificing efficiency. Second, search procedures directly supported by language abstractions, i.e., nondeterminism in logic programming, must be available in traditional languages and remain under end-user control. Also, search procedures should retain their declarative nature and style to preserve simplicity and appeal. Finally, a constraint tool must bridge the semantic gaps that exists between a high-level model, its implementation and the native abstractions of the host language to preserve clarity and simplicity while offering a natural embedding in the target language that does not force its users to become logic programming experts.

Many answers to these challenges have been offered and strike different trade-offs. Each answer can be characterized by features that address a subset of the objectives. Some constraint tools favored ease of adoption and efficiency over preserving declarativeness and flexibility. Others focused on the creation of new *hybrid*, multi-paradigm languages and platforms that preserved declarative constructions, adopt constraints and concurrency as first class citizens in the design, and preserve efficiency with a lesser emphasis on targeting existing platforms. A third option focusing on flexibility and declarative constructions brought rule-based systems where new solvers over entirely new domains can be easily constructed, extended, modified and composed.

This chapter provides insights into the strengths, weaknesses and capabilities of systems that fall in one of these three classes: toolkits for procedural and object-oriented languages, hybrid systems and rule-based systems.

1.1 Procedural and Object-Oriented Languages

Over the last decade, constraint programming tools have progressively found their way into mainstream paradigms and languages, most notably C++. This transformation, however, is not obvious and creates many challenges. To understand the nature of the difficulty, it is useful to step back and consider the initial motivations. To apprehend the interactions between constraint toolkits and their procedural or object-oriented host languages, it is useful to separate two key components of constraint programming, i.e., *modeling with constraints* and *programming the search*. Each component brings its own challenges that vary with the nature of the host language. Section 1.1.1 reviews the design objectives and inherent challenges before turning to the issue of constraint-based modeling in section 1.1.2, and search programming in section 1.1.3. Finally, section 1.1.4 discusses pragmatic issues that permeate throughout all integration attempts.

1.1.1 Design Objectives

Modern constraint-based languages strive to simplify the task of writing models that are readable, flexible and easy to maintain. This is naturally challenging as programs for complex problems often requires ingenuity on the part of the developer to express multiple orthogonal concerns and encode them efficiently within a language that imposes its own

limitations. Logic programming is the cradle of constraint programming for good reasons as it offers two important supports: a declarative framework on which to build constraints as generalizations of unification; and non-determinism to support search procedures, and, in particular, depth-first search.

The Challenges

Nonetheless, logic programming imposes a few limitations. First, it does not lend itself to the efficient implementation of extensible toolkits. Early on, efficiency considerations as well as simplicity pushed logic programming systems to implement all constraints as "built-ins" of the language giving rise to closed *black-box* solvers. Second, it does not easily accommodate search procedures that deviate from the depth-first strategy. It therefore raises significant challenges for potential users of strategies like BFS, IDFS [69] or LDS [50] to name a few. Third, its target audience comprises almost exclusively computer-savvy programmers, who feel comfortable writing recursive predicates for tasks as mundane as generating constraints and constraint expressions. This relative difficulty does not appeal to a much larger group of potential users of the technology: the mathematical programming community. Mathematical solvers (LP, IP, MIP [55, 46]) and their modeling languages [34, 26, 103] indeed offer facilities that focus on modeling and, to a large extent, relieve their users from most (all) programming efforts.

The past two decades saw improvements on one or more of these fronts. The next paragraphs briefly review two trends *related to* procedural and object-oriented languages.

Libraries and glass-box extensibility. Ilog Solver [57] is a C++ library implementing a finite domain solver and is thus a natural example of an object-oriented embodiment. The embedding of a solver within a C++ library offers opportunities to address the extensibility issues as both decision variables and constraints can be represented with object hierarchies that can be refined and extended. However the move to C++, a language that does not support non-deterministic computation, has increased the challenges one faces to write, debug and maintain search procedures. Note that CHR [39] supports glass-box extensibility through user-definable rules and is the subject of Section 1.3.

From programming to modeling. Numerica [110] is a modeling language for highly non-linear global optimization. It was designed to address the third limitation and make the technology of Newton [111] (a constraint logic programming language) available to a much broader audience of mathematical programmers. The objective behind Numerica was to improve the modeling language to a point where executable models were expressed at the level of abstraction of their formal counterparts typically found in scientific papers. The approach was further broadened with novel modeling languages for finite domain solvers supporting not only the statement of constraints but also the specification of advanced search procedures and strategies. OPL [104, 103, 113] embodies those ideas in a rich declarative language while OplScript [107] implements a procedural language for the composition of multiple OPL models. Note that OPL is an interpreted language whose virtual machine is implemented in terms of ILOG SOLVER constructions. The virtual machine itself is non trivial given the semantic gap between OPL and ILOG SOLVER.

At the same time, a finite domain solver was implemented in LAURE [21] and then moved to CLAIRE [62], a language compiled to C++ that simplified LAURE's construc-

tions to make it accessible to a broader class of potential users. CLAIRE was later enhanced with SALSA [62], a declarative and algebraic extension that focused on the implementation of search procedures.

1.1.2 Constraint Modeling

Constraint modeling raises two concerns: the ease of use and expressiveness of the toolkit and its underlying extensibility. Each concern is intrinsically linked to the host language and has a direct impact on potential end users. This section discusses each one in turn.

Ease of Use and Expressiveness

The constraint modeling task within a procedural or an object-oriented language presents interesting challenges. It is desirable to obtain a *declarative* reading of a *high-level* model statement that exploits the facilities of the host language (e.g., static and strong typing in C++). The difficulty is to leverage the language to simplify programs and raise their modeling profile to a sufficient level of abstraction. Note that modeling languages (e.g., OPL) tightly couple the toolkit and the language to obtain the finest level of integration that preserves a complete declarative reading of models despite an apparent procedural style. Indeed, OPL looks procedural but is actually declarative as it is side effect free (i.e., it has no destructive assignments).

Aggregation and combinators. Consider the classic magic series puzzle. The problem consists of finding a sequence of numbers $S = (s_0, s_1, \cdots, s_{n-1})$ such that s_i represents the number of occurrences of i within the sequence S. For instance, $(6, 2, 1, 0, 0, 0, 1, 0, 0, 0)$ is a sequence of length 10 with 6 occurrences of 0, 2 occurrences of 1, 1 occurrence of 2 and finally 1 occurrence of 6. Clearly, any solution must satisfy the following property

$$\sum_{k=0}^{n-1}(s_k = i) = s_i \;\; \forall i \in \{0, 1, 2, \cdots, n-1\}$$

To solve the problem with a constraint programming toolkit, it is first necessary to state the n constraints shown above. Each constraint is a linear combination of the truth value (interpreted as 0 or 1) of elementary constraints of the form $s_k = i$. The difficulty is therefore to construct a toolkit with automatic reification of constraints and with seamless aggregation primitives, i.e., summations, products, conjunctions or disjunctions to name a few that facilitate the combination of elementary primitives.

Figure 1.1 illustrates the differences between the OPL and ILOG SOLVER statements for the magic series problem. The ILOG SOLVER model constructs an expression iteratively to build the cardinality constraint for each possible value. It also relies on convenience functions like `IloScalProd` to create the linear redundant constraint. The OPL model is comparatively simpler as the mathematical statement maps directly to the model. It is worth noting that the level of abstraction shown by the OPL model is achievable within C++ libraries with the same level of typing safety as demonstrated in [71]. Finally, both systems implement constraint combinators (e.g., cardinality) and offer global constraints that capture common substructures, simplify some of the modeling effort, and can exploit the semantics of constraints for better performance.

ILOG SOLVER

```
1. int main(int argc,char* argv[]){
2.   IloEnv env;int n;cin>>n;
3.   try {
4.     IloModel m(env);
5.     IloIntVarArray s(env,n,0,n);
6.     IloIntArray c(env,n);
7.     for(int i=0;i<n;i++) {
8.       IlcIntExp e = s[0] == i;
9.       for(int k=1;k<n;k++)
10.        e += s[k] == i;
11.      m.add(s[i] == e);
12.    }
13.    for(int i=0;i<n;i++) c[i]=i;
14.    m.add(IloScalProd(s,c) == n);
15.    solve(m,env,vars);
16.  } catch(IloException& ex) ...
17.}
```

OPL

```
1. int n<<"Number of Variables:";
2. range Dom 0..n-1;
3. var Range s[Dom];
4. solve {
5.   forall(i in Dom)
6.     s[i] = sum(j in Dom) (s[j]=i);
7.   sum(j in Dom) s[j]*j = n;
8. };
```

Figure 1.1: The Magic Series statements.

Typing. A seamless toolkit integration depends on the adherence to the precepts and conventions of the host language. For instance, C++ programmers often expect static and strong typing for their programs and rely on the C++ compiler to catch mistakes through type checking. From a modeling point of view the ability to rely on types and, in particular, on finite domain variables defined over domains of specific types is instrumental is writing clean and simple models. Consider the stable marriage problem. The problem is to pair up men and women such that the pairings form marriages and satisfy stability constraints based on the preferences of all individuals. A marriage between m and w is stable if and only if whenever m prefers a woman k over his wife w, k also happens to prefer her own husband over m so that m and w have no reason to part. The OPL model is shown in Figure 1.2. The fragment `husband[wife[m]] = m` illustrates that the type of values in the domain of `wife[m]` is an enumerated type `Women` that happens to be equal to the type of the index for the array `husband`. Similarly, the type of each entry of the husband array is `Men` and therefore equal to the type of the right hand side of the equality constraint. To the modeler, the result is a program that can be statically type checked.

Matrices. From an expressiveness point of view, the ability to index arrays with finite domain variables is invaluable to write concise and elegant models. It is equally useful on matrices, especially when its absence implies a non trivial reformulation effort to derive for an expression $m[x, y]$ a *tight* reformulation based on an element constraint. The reformulation introduces a ternary relation $R(i, j, k) = \{\langle i, j, k\rangle \mid i \in D(x) \wedge j \in D(y) \wedge k \in 0..|D(x)| \cdot |D(y)| - 1\}$ that, for each pair of indices $i \in D(x)$ and $j \in D(y)$, maps the entry $m[i, j]$ to its location k in an array a. Then, $m[x, y]$ can be rewritten as $a[z]$ with the addition of the constraint $(x, y, z) \in R$ where z is a fresh variable.

Note that if the language supports a rich parametric type system (e.g., C++), it is possible to write templated libraries that offer both automatic reformulations and static/strong typing as shown in [71].

```
1.  enum Women ...;
2.  enum Men ...;
3.  int rankW[Women,Men] = ...
4.  int rankM[Men,Women] = ...
5.  var Women wife[Men];
6.  var Men husband[Women];
7.  solve {
8.    forall(m in Men)    husband[wife[m]] = m;
9.    forall(w in Women)  wife[husband[w]] = w;
10.   forall(m in Men & o in Women)
11.     rankM[m,o] < rankM[m,wife[m]] => rankW[o,husband[o]] < rankW[o,m];
12.   forall(w in Women & o in Men)
13.     rankW[w,o] < rankW[w,husband[w]] => rankM[o,wife[o]] < rankM[o,w];
14. }
```

Figure 1.2: The OPL model for Stable Marriage.

Extensibility

Extensibility is crucial to the success of toolkits and libraries alike. It affects them in at least two respects. First, the toolkit or library itself should be extensible and support the addition of user-defined constraints and user-defined search procedures. This requirement is vital to easily develop domain specific or application specific constraints and blend them seamlessly with other pre-defined constraints. Given that constraints are compositional and implemented in terms of filtering algorithms that task should be easily handled. Second, it is often desirable to embed the entire constraint program within a larger application to facilitate its deployment.

```
1.  class MyEqual :  public IlcConstraintI {
2.    IlcIntVar _x,_y;
3.  public:
4.    MyEqual(IloSolver s,IlcIntVar x,IlcIntVar y)
5.      :  IlcConstraintI(s),_x(x),_y(y) {}
6.    void post() {
7.      _x.whenValue(equalDemon(getSolver(),this,_x));
8.      _y.whenValue(equalDemon(getSolver(),this,_y));
9.    }
10.   void demon(IlcIntVar x) {
11.     IlcIntVar other = (x == _x) ?  _y :  _x;
12.     other.setMin(x.getMin());
13.     other.setMax(x.getMax());
14.   }
15. };
16. ILCCTDEMON1(equalDemon,MyEqual,demon,IlcIntVar,var);
```

Figure 1.3: ILOG SOLVER custom constraint.

Solver extensibility. Object orientation is a paradigm for writing extensible software through a combination of polymorphism, inheritance, and delegation. In the mid 90s, the first version of ILOG SOLVER [78, 79] was developed to deliver an extensible C++ library.

The extensibility of its modeling component stems from a reliance on abstract classes (interfaces) for constraints to specify the API that must be supported to react to events produced by variables. For instance an ILOG SOLVER integer variable can expose notifications for three events `whenDomain, whenRange, whenValue` to report a change in the domain, the bounds, or the loss of a value. A constraint subscribes to notifications from specific variables to respond with its `demon` method. Figure 1.3 illustrates a user-defined equality constraint implementing bound consistency. Its `post` method creates two demons and attaches them to the variables. Both demons are implemented with a macro (last line) that delegates the event back to the constraint. The `demon` method propagates the constraint by updating the bound of the other variable. The extension mechanism heavily depends on the specification of a *filtering algorithm* rather than a set of *indexicals* (e.g., clp(FD) [28]) or *inference rules* (e.g., CHR [39]) and therefore follows a far more procedural mind-set that requires a fair level of understanding to identify relevant events and variables and produce a filtering procedure.

Solver embedding. Extensibility also matters for the deployment of constraint-based technology. In this respect, the integration of a CP toolkit within a mainstream object-oriented language is a clear advantage as models can easily be encapsulated within reusable classes linked within larger applications. Modeling languages present an additional difficulty but can nonetheless be integrated through component technology (COM or CORBA) [56] or even as web-services as illustrated by the OSiL efforts [33].

1.1.3 Programming the Search

The second component of a constraint programming model is concerned with the search. The search usually addresses two orthogonal concerns. First, *what* is the topology of the search tree that is to be explored. Second, *how* does one select the next node of the search tree to be explored. Or, given a search tree, what is the order used to visit its nodes? Both can be thought of as declarative specifications but are often mixed to accommodate the implementation language. The integration of the two elements in procedural and object-oriented languages is particularly challenging, given the lack of language abstractions to manipulate the search control flow.

Search Tree Specification

OPL is a classic example of declarative specification of the search tree. It supports statements that specify the order in which variables and values must be considered. OPL provides default strategies and does not require the user to implement his own. However, as problems become more complex, it is critical to provide this ability. Figure 1.4 illustrates on the left-hand side the naive formulation for the $n-$queens model. The constraints are stated for all pairs of indices i and j in `Dom` such that $i < j$. The right-hand side shows the search procedure. Lines 10-14 specify the search tree with a variable and a value ordering . It simply scans the variables in the order indicated by `Dom` (ascending) and, for each variable i, it non-deterministically chooses a values v from `Dom` and attempts to impose the additional constraint `queen[i] = v`. On failure, the non-deterministic choice is reconsidered and the next value from `Dom` is selected.

```
1.  int n = ...; range Dom 1..n;
2.  var Dom queen[Dom];
3.  solve {
4.    forall(ordered i,j in Dom){
5.      queen[i] <> queen[j];
6.      queen[i]+i <> queen[j]+j;
7.      queen[i]-i <> queen[j]-j;
8.    }
9.  }
10. search {
11.   forall(i in Dom)
12.     tryall(v in Dom)
13.       queen[i] = v;
14. }
```

Figure 1.4: The OPL queens model.

Implementing a search facility in an object-oriented language like C++ or Java is hard for a simple reason: the underlying language has no support for non-determinism and therefore no control abstractions for making choices like `tryall`. To date, all libraries have used some form of embedded *goal interpreter* whose purpose is to evaluate an *and-or tree* data structure reminiscent of logic programming predicates where non-determinism is expressed with or-nodes and conjunction with and-nodes. The approach was used in ILOG SOLVER and more recently in CHOCO, a Java-based toolkit. Figure 1.5 shows a goal-based implementation of the $n-$queens search tree. ILOG SOLVER also provides predefined search tree specifications for the often-used methods.

```
1.  ILCGOAL4(Forall,IloIntVarArray,x,IloInt,i,IloInt,low,IloInt,up) {
2.    if (i <= up)
3.      return IlcAnd(Tryall(getSolver(),x[i],low,up),
4.                    Forall(getSolver(),x,i+1,low,up));
5.    else return IlcGoalTrue(getSolver());
6.  }
7.  ILCGOAL3(Tryall,IloIntVar,x,IloInt,v,IloInt up) {
8.    if (x.isBound()) return 0;
9.    else if (v > up) fail();
10.   else return IlcOr(x=v,IlcAnd(x!=v,Tryall(getSolver(),x,v+1)));
11. }
12. ...
13. solver.solve(Forall(queens,1,1,n));
```

Figure 1.5: An ILOG SOLVER implementation of a search tree specification.

Lines 1 through 6 define a goal that performs the same variable selection as line 11 of the OPL model. Lines 7-11 define a goal to try all the possible values for the chosen variable and correspond to lines 12 and 13 of the OPL model. The two macros `ILCGOAL4` and `ILCGOAL3` define two classes (`ForallI` and `TryallI`) together with convenience functions (`Forall` and `Tryall`) to instantiate them[1]. The block that follows each macro is the body of the goal whose purpose is to construct the And-Or tree on the fly.

Observe that the implementation of the search procedure is now broken down into several small elements that are not textually close. A few observation are in order

- A goal-based solution relies on an embedded goal interpreter and is therefore incompatible with C++ development tools like a debugger. For instance, tracing the

[1]Observe that the code in Figure 1.5 always uses the convenience functions and never directly refer to the underlying implementation class

execution is hard as there is no access to the state of the interpreter (e.g., current instruction, parameters' value, etc..). To compensate, recent versions of ILOG SOLVER provide debugging support through instrumented libraries to inspect and visualize the state of the search tree.

- Every single operation that must occur during the search (e.g., printing, statistic gathering, visualizations) must be wrapped up in user-defined goals that are inserted into the search tree description.

- It is non-trivial to modularize entire search procedures in *actual C++ functions or classes* to reuse search fragments. Again, the only option is to write a function or class that will *inline* a goal data structure representing the search procedure to insert. Note that a deep copy of the entire goal (the entire function) is required each time to simulate the parameter passing as there is no call mechanism per se.

- The body of a goal's implementation is both delicate and subtle as there is a temporal disconnection between the execution of its various components. For instance, one may be tempted to optimize the `Forall` goal shown in Figure 1.5 to eliminate the creation of a fresh goal instance for each recursive goal and favor a purely recursive solution as in

```
ILCGOAL4(Forall,IloIntVarArray,x,IloInt,i,IloInt,low,IloInt,up) {
    if (i <= up) {
      IloInt i0 = i; i = i + 1;
      return IlcAnd(Tryall(getSolver(),x[i0],low,up),this);
    } else return IlcGoalTrue(getSolver());
}
```

  However, this would be wrong. Indeed, i is an instance variable of the goal that is merely re-inserting itself back into the query resulting in making $i = i + 1$ visible to the next invocation. However, on backtrack i is *not* restored to its original value. Consequently, one must compensate wit a *reversible* integer (`IlcRevInt`). Yet, this is insufficient as the modification ($i = i + 1$) should occur *inside* the `Tryall` choice point and it is thus necessary to add a goal to increment i as in

```
ILCGOAL4(Forall,IloIntVarArray,x,IlcRevInt&,i,IloInt,low,IloInt,up) {
    if (i <= up) {
      return IlcAnd(IlcAnd(Tryall(getSolver(),x[i],low,up),
                           IncrementIt(i)),this);
    } else return IlcGoalTrue(getSolver());
}
```

  Finally, note how the arguments to goal instantiations are evaluated when the parent goal executes, *not* when the goal itself is about to execute. For instance, a goal that follows `IncrementIt(i)` should not expect i to be incremented yet.

Standard search procedures are not limited to static variable/value ordering but often rely on dynamic heuristics in order to select the next variable/value to branch on. Such heuristics can be implemented both within modeling languages and libraries.

Variable selection heuristic. In OPL, the variable selection heuristic is specified with a clause in the `forall` statement that associates with the selection a measure of how desirable the choice is. For instance, the fragment

```
forall(i in Dom ordered by increasing dsize(queens[i])) ...
```

indicates that the queens should be tried in increasing order of domain size. Note that OPL supports more advanced criteria based on lexicographic ordering of tuple-values to automate a useful but tedious task. For instance, the fragment

```
forall(i in Dom ordered by increasing <dsize(queens[i]),abs(i - n/2)>)
    tryall(v in Dom)
      queen[i] = v;
```

implements a middle variable selection heuristic that considers first the variable with the smallest domain and breaks ties by choosing the variable closest to the middle of the board.

ILOG SOLVER is equally capable at the expense of a few small additions to user-defined goals. Indeed, the key change is that the index of the next variable to consider is no longer a static expression (the i of the `Forall` goal in Figure 1.5), but is instead computed at the beginning of the goal. Note that the selection is re-done at each invocation of the `Forall` and can skip over bound variables.

Value selection heuristic. OPL provides an ordering clause for its `tryall` that matches the variable ordering clause of the `forall` both in syntax and semantics. For instance the statement

```
tryall(v in Domain ordered by increasing abs(v - n/2)) ...
```

would consider the values from $Domain$ in order of increasing distance from the middle of the board. ILOG SOLVER goals for the value selection operate similarly with one caveat: The value selection goal must track (with an additional data structure) the already tried values to focus on only the remaining values, a task hidden by OPL's implementation.

Control flow primitives. For the search, the most significant difference between a modeling language and a library is, perhaps, the availability of traditional control statements. As pointed out earlier, ILOG SOLVER's level of abstraction for programming the search is the underlying and-or tree. OPL, provides traditional control primitives such as iterations (`while` loops), selections (`select`), local bindings (`let` expressions) and branchings (`if-then-else`). Consider for instance the simple OPL fragment shown in Figure 1.6 which, upon failure, adds the negation of the failed constraint. The distance between a goal-based specification and a high-level procedure is significant.

```
1.  search {
2.    forall(in in Dom)
3.      while (not bound(queens[i])) do
4.        let v = dmin(queens[i]) in
5.          try
6.            queens[i] = v | queens[i] <> v
7.          endtry;
8.  }
```

Figure 1.6: Traditional Control Abstractions Example in OPL.

Exploration Strategies

The specification of the search tree was concerned with *what* was going to be explored. Exploration strategies are concerned with *how* the dynamic search tree is going to be explored. Many strategies are possible, ranging from the standard depth first search to complex combination of iterated limited searches. Even though an exploration strategy sounds like a very algorithmic endeavor, it is both possible and desirable to produce a declarative specification and let the search engine implement it automatically. This is especially true in the context of a procedural (or object-oriented) language as a procedural specification would force programmers to explicitly address the issue of non-determinism (and its implementation). This section briefly reviews two approaches based on OPL [112] (or ILOG SOLVER [77]) and COMET [72].

OPL and ILOG SOLVER strategy specifications. The key ingredient to specify an exploration strategy is to provide a search node management policy. Each time a choice is considered during the search, it creates search nodes corresponding to the various alternatives. Once created, the exploration must *select* the node to explore next and *postpone* the less attractive ones. The *evaluation* of a node's attractiveness is, of course, strategy dependent. But once the attractiveness function and the postponement rules are encaspulated in a strategy object, the exploration algorithm becomes completely generic with respect to the strategy.

```
1.SearchStrategy dfs() {
2.  evaluated to - OplSystem.getDepth();
3.  postponed when OplSystem.getEvaluation()>OplSystem.getBestEvaluation();
4.}

5.applyStrategy dfs()
6.  forall(i in Dom)
7.    tryall(v in Dom)
8.      queen[i] = v;
```

Figure 1.7: Exploration Strategy in OPL.

Consider the statement in Figure 1.7. It first defines a DFS strategy and uses it to explore the search tree. The specification contains two elements: the evaluation function that defines the node's attractiveness and the postponement rule that states when to delay. Each time the exploration produces a node, it is subjected to the strategy to evaluate its attractiveness and decide its fate. To obtain DFS, it suffices to use the opposite of the node's depth as its attractiveness and to postpone a node whenever it is shallower than the "best node" available in the queue. The system object (`OplSystem` gives access to enough statistic about the depth, right depth, number of failures, etc..) to implement advanced strategies like LDS or IDS to name a few. When the strategy is expressed as a node management policy, one can implement the same mechanism in a library.

COMET strategy specifications. COMET [70] is an object-oriented programming language for constraint-based local search offering control abstractions for non-determinism [105].

These abstractions are equally suitable for local search methods (low overhead) and complete methods.

COMET uses first-class continuations to represent and manipulate the state of the program's control flow. COMET's `tryall` is semantically equivalent to OPL's `tryall`. Search strategies can be expressed via policies for the management of the captured continuations and embedded in `Search Controllers` that parameterize the search.

```
1. DFS sc();
2. exploreall<sc> {
3.   forall(i in Dom) {
4.     tryall<sc>(v in Dom){
5.       queen[i] = v;
6.     }
7.   }
8. }
```

```
1. class DFS implements SearchController {
2.   Stack _s; Continuation _exit;
3.   DFS() { _s = new Stack();}
4.   void start(Continuation c) {_exit = c;}
5.   void exit() { call(_exit);}
6.   void addChoice(Continuation c) {
7.     _s.push(c);
8.   }
9.   void fail() {
10.    if (_s.empty()) exit();
11.    else call(_cont.pop());
12.  }
13.}
```

Figure 1.8: Exploration strategies with COMET.

The code fragment on the left hand side of Figure 1.8 is a COMET procedure whose semantics are identical to the OPL statement from Figure 1.7. The key difference is the search controller (`sc`) of type `DFS` whose implementation is shown on the right hand side. The statements parameterized by `sc` (`exploreall` and `tryall`) delegate to the search controller the management of the continuations that represent search nodes. To derive DFS, it suffices to store in a stack the continuations produced by the branches in the `tryall`. When a failure occurs (e.g., at an inconsistent node), the fail method transfers the control to the popped continuation. If there is none left, the execution resumes after the `exploreall` thanks to a call to the `_exit` continuation.

COMET completely decouples the node management policy from the exploration algorithm, allows both a declarative and an operational reading of the search specification and provides a representation of the control flow's state that is independent of the underlying computation model.

1.1.4 Pragmatics

The integration of a constraint programming toolkit within a purely procedural or object-oriented language presents challenges for the modeling and implementation of the search.

Constraint Modeling

Constraint modeling is relatively easy *if* the host language supports first-class expressions or syntactic sugar to simulate them. If operators cannot be overloaded (like in Java), the expression of arithmetic and set-based constraint is heavier. See Figure 1.9 for a Java fragment setting up the queens problem in the CHOCO solver.

```
1.  Problem p = new Problem();
2.  IntVar[] queens = new IntVar[n];
3.  for(int i = 0; i < n; i++)
4.    queens[i] = p.makeEnumIntVar("queen" + i, 1, n);
5.  for (int i=0; i<n; i++) {
6.    for (int j=i+1; j<n; j++) {
7.      p.post(p.neq(queens[i], queens[j]));
8.      p.post(p.neq(queens[i],p.plus(queens[j], j-i)));
9.      p.post(p.neq(queens[i],p.minus(queens[j], j-i)));
10.   }
11. }
```

Figure 1.9: The $n-$queens problem in CHOCO.

Search Implementation

The lack of support for non-determinism is far more disruptive. One extreme solution is to close the specification of the search and only offer pre-defined procedures. The clear advantage is an implementation of non-determinism that can be specialized to deliver good performance.

A second option, used with ILOG SOLVER [57], is to embed in the library a goal oriented interpreter. With a carefully crafted API addressing the issues listed below, it is possible to open the interface to support user-defined extensions.

Control transfer. The interface between the goal-based search and the rest of the program must be as seamless as possible.

Mixed memory models. Multiple memory models must coexist peacefully (traditional C Heap, logical variables Heap, traditional execution stack, search stack or trail to name a few) to avoid leaks or dangling pointer issues.

Debugging support. A significant part of the program runs inside an embedded interpreter which renders the native debugging facilities virtually useless. This must be mitigated with the inclusion of dedicated and orthogonal debugging tools to instrument the goal interpreter.

Control abstractions. The native control abstraction tend to be ineffective to express search procedures and underscore the importance of hiding or isolating the semantic subtleties associated with the goal interpreter. Note that the level of abstraction of search procedures can be lifted closer to OPL as demonstrated in [71]. However, this implementation retains a goal-like interpreter that also fails to integrate with existing tools.

1.2 Concurrent Constraint Programming

At the end of the 1980s, concurrent constraint logic programming (CCLP) integrated ideas from concurrent logic programming [96] and constraint logic programming (CLP):

- Maher [64] proposed the ALPS class of committed-choice languages.

- The ambitious Japanese Fifth-Generation Computing Project relied on a concurrent logic language based on Ueda's GHC [102].

- The seminal work of Saraswat [81] introduced the *ask-and-tell* metaphor for constraint operations and the concurrent constraints (CC) language framework that permits both don't-care and don't-know non-determinism.

- Smolka proposed a concurrent programming model Oz that subsumes functional and object-oriented programming [100].

Implemented concurrent constraint logic programming languages include AKL, CIAO, CHR, and Mozart (as an implementation of Oz).

1.2.1 Design Objectives

Processes are the main notion in concurrent and distributed programming. They are building blocks of *distributed systems*, where data and computations are physically distributed over a network of computers. *Processes* are programs that are executed concurrently and that can interact with each other. Processes can either execute local actions or *communicate* and *synchronize* by sending and receiving messages. The communicating processes build a *process network* which can change dynamically. For concurrency it does not matter if the processes are executed physically in parallel or if they are interleaved sequentially. Processes can intentionally be non-terminating. Consider an operating system which should keep on running or a monitoring and control program which continuously processes incoming measurements and periodically returns intermediate results or raises an alarm.

In CCLP, concurrently executing processes communicate via a shared constraint store. The processes are defined by predicates and are called *agents*, because they are defined by logical rules and often implement some kind of artificially intelligent behavior. Constraints take the role of (partial) messages and variables take the role of communication channels. Usually, communication is asynchronous. Running processes are CCLP goals that place and check constraints on shared variables.

This communication mechanism is based on *ask-and-tell* of constraints that reside in the common constraint store. *Tell* refers to imposing a constraint (as in CLP). Ask is an inquiry whether a constraint already holds. *Ask* is realized by an *entailment* test. It checks whether a constraint is implied by the current constraint store. Ask and tell can be seen as generalizations of read and write from values to constraints. The ask operation is a *consumer* of constraints (even though the constraint will not be removed), the tell operation is a *producer* of constraints.

For a process, decisions that have been communicated to the outside and actions that have affected the environment cannot be undone anymore. *Don't-know non-determinism* (Search) must be encapsulated in this context. Also, failure should be avoided. Failure of a goal atom (i.e., a single process) always entails the failure of the entire computation (i.e., all participating processes). In applications such as operating or monitoring systems this would be fatal.

1.2.2 The CC Language Framework

We concentrate on the committed-choice fragment of Saraswat's CC language framework [82, 83, 80]. The abstract syntax of CC is given by the following EBNF grammar:

$$\begin{array}{lll} \text{Declarations} & D ::= & p(\tilde{t}) \leftarrow A \mid D, D \\ \text{Agents} & A ::= & \mathit{true} \mid tell(c) \mid \sum_{i=1}^{n} ask(c_i) \rightarrow A_i \mid A \| A \mid \exists x p(\tilde{t}) \mid p(\tilde{t}) \end{array}$$

where $\tilde{t}$ stands for a sequence of terms, x for a variable, and where c and the c_i's are constraints. Instead of using existential quantification ($\exists$), projection is usually implicit in implemented CC languages by using local variables as in CLP.

Each predicate symbol p is defined by exactly one declaration. A *CC program* P is a finite set of declarations.

The operational model of CC is described by a transition system. States are pairs consisting of agents and the common constraint store. The transition relation is given by the transition rules in Fig. 1.10.

Tell	$\langle tell(c), d\rangle \rightarrow \langle \mathit{true}, c \wedge d\rangle$	
Ask	$\langle \sum_{i=1}^{n} ask(c_i) \rightarrow A_i, d\rangle \rightarrow \langle A_j, d\rangle$	if $CT \models d \rightarrow c_j$ $(1 \leq j \leq n)$
Composition	$\dfrac{\langle A, c\rangle \rightarrow \langle A', c'\rangle}{\begin{array}{l}\langle (A \parallel B), c\rangle \rightarrow \langle (A' \parallel B), c'\rangle \\ \langle (B \parallel A), c\rangle \rightarrow \langle (B \parallel A'), c'\rangle\end{array}}$	
Unfold	$\langle p(\tilde{t}), c\rangle \rightarrow \langle A \parallel tell(\tilde{t} = \tilde{s}), c\rangle$	if $(p(\tilde{s}) \leftarrow A)$ in program P

Figure 1.10: CC transition rules

Tell $tell(c)$ adds the constraint c to the common constraint store. The constraint *true* always holds.

Ask *Don't care non-determinism* between choices is expressed as $\sum_{i=1}^{n} ask(c_i) \rightarrow A_i$. One nondeterministically chooses one c_i which is implied by the current constraint store d, and continues computation with A_i.

Composition The $\parallel$ operator enables parallel composition of agents. Logically, it is interpreted as conjunction.

Unfold Unfolding replaces an agent $p(\tilde{t})$ by its definition according to its declaration.

A finite CC derivation (computation) can be successful, failed or deadlocked depending on its final state. If the derivation ends in a state with unsatisfiable constraints it is called *failed*. Otherwise, the constraints of the final state are satisfiable. If its agents have reduced to *true*, then it is *successful*, else it is *deadlocked* (i.e., the first component contains at least one suspended agent). Deadlocks come with concurrency. They are usually considered programming errors or indicate a lack of sufficient information to continue computation.

1.2.3 Oz and AKL as Concurrent Constraint Programming Languages

The concurrent constraint programming model establishes a clean and simple model for synchronizing concurrent computations based on constraints. On the other hand, CLP (see Chapter 12, "Constraint Logic Programming") provides support for modeling and solving combinatorial problems based on constraints. The obvious idea to integrate both models to yield a single and uniform model for concurrent and parallel programming and problem solving however has proven itself as challenging. Besides merging concurrency and problem solving aspects, the CCP model only captures synchronization based on a single shared constraint store. Other common aspects such as controlling the amount of concurrency in program execution and exchanging messages between concurrently running computations are not dealt with.

These challenges and issues have been one main motivation for the development of AKL and Oz as uniform programming models taking inspirations from both CCP and CLP. The development of AKL started before that of Oz, and naturally Oz has been inspired by many ideas comning from AKL. Later, the two development teams joined forces to further develop Oz and its accompanying programming system Mozart [76]. As Oz integrates all essential ideas but parallel execution from AKL, this section puts its focus on Oz and mentions where important ideas have been integrated from AKL. Achieving parallelism has been an additional motivation in AKL, this resulted in a parallel implementation of AKL [75, 74].

Currently Oz and Mozart are used in many different application areas where the tight combination of concurrency and problem solving capabilities has shown great potential. Education is one particular area where many different programming paradigms can be studied in a single language [114]. Oz as a multi-paradigm language is discussed in [116].

1.2.4 Expressive Concurrent Programming

The concurrent constraint programming model does not specify which amount of concurrency is necessary or useful for program execution. This is clearly not practical: the amount of concurrency used in program execution makes a huge difference in efficiency. The rationale is to use as little concurrency as possible and as much concurrency as necessary.

Experiments with Oz for the right amount of concurrency range from an early ultra-concurrent model [52], over a model with implicit concurrency control [99] to the final model with explicit concurrency control. Explicit concurrency control means that execution is organized into threads that are explicitly created by the programmer. Synchronization then is performed on the level of threads rather than on the level of agents as in the CCP model.

Many-to-one communication. Variables in concurrent constraint programming offer an elegant mechanism for one-to-many communication: a variable serves as a communication channel. Providing more information on that variable by a tell amounts to message sending on that variable. The variable then can be read by many agents with synchronization through entailment on the arrival of the message.

With constraints that can express lists (such as constraints over trees) programs can easily construct streams (often referred to as open-ended lists). A stream is defined by a

current tail being a yet unconstrained variable t. Sending a message m tells the constraint $t = \mathrm{cons}(m, t')$ (expressing that the message m is the first element of the stream t followed by elements on the stream t') where t' is a new variable for the new current tail of the stream.

This idea for stream-based communication is very useful for programming concurrent applications [96, 81]. However, it has a serious shortcoming: it does not support many-to-one communication situations where more than a single sender exists. The tail can be only constrained at most once by a tell. Hence all potential senders need to know and update the current tail of a stream.

AKL introduced *ports* to solve this problem and allow for general message-passing communication [61]. The importance of supporting general message-passing communication is witnessed by concurrent programming languages where communication is entirely based on message passing, for example Erlang [13].

A port provides a single point of reference to a stream of messages. It stores the current tail of the stream that is associated with a port. Ports provide a send operation. The send operation takes a port and a message, appends the message to the tail of the port's stream, and updates the stream's tail as described above.

Naming entities. Ports in AKL require that they can be referred to for a send operation. Modeling a port as a constraint in the concurrent constraint programming framework is impossible. The very idea of a port is that its associated tail changes with each send operation. Changing the tail is in conflict with a monotonically growing constraint store.

A generic solution to this problem has been conceived in Oz by the introduction of *names* [99]. A name can be used similar to a constant in a constraint. Additionally, the state of a computation now also has an additional compartment that maps names to entities (such as ports). For example, using a name n for a port means that constraints can be used to refer to the port by using the name n. The additional compartment then stores that n refers to a port and the current tail associated with that port. Names are provided in a way that they cannot be forged and are unique, more details are available in [99].

Mutable state. Ports are not primitive in Oz. Ports are reduced to cells as a primitive that captures mutable state. As discussed above, a cell is referred to by a name and the only operation on a cell is to exchange its content. From cells, ports can be obtained straightforwardly [100].

More expressive programming. Oz incorporates extensions to the concurrent constraint model to increase its expressive power for programming. It adds first-class procedures by using names to refer to procedural abstractions (closures). By this, the aspect of giving procedures first-class status is separated from treating them in the underlying constraint system. The constraint system is only concerned with names referring to procedural abstractions but not with their denotation. This approach also supports functional computation by simple syntactic transformations [100].

The combination of names, first-class procedures, and cells for mutable state constitute the ingredients necessary for object-oriented computing. Here names are used as references to objects, mutable object state is expressed from cells, and classes are composed

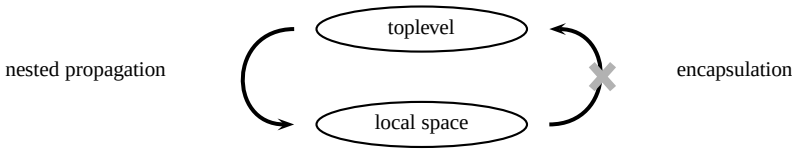


Figure 1.11: Nested propagation and encapsulation for spaces.

from first-class procedures. This setup allows for full-fledged concurrent object-oriented programming including object-based synchronization and class-based inheritance [51].

Distributed programming. The basic idea of distribution in Oz is to abstract away the network as much as possible. This means that all network operations are invoked implicitly by the system as an incidental result of using particular language operations. Distributed Oz has the same language semantics as Oz by defining a distributed semantics for all language entities such as variables or objects based on cells.

Network transparency means that computations behave the same independent of the site they compute on, and that the possible interconnections between two computations do not depend on whether they execute on the same or on different sites. Network transparency is guaranteed in Distributed Oz for most entities.

An overview on the design of Distributed Oz is [48]. The distributed semantics of variables is reported in [49]; the distributed semantics of objects is discussed in [115].

1.2.5 Encapsulation and Search

The main challenge in combining concurrency with problem solving is that constraint-based computations used for problem solving are *speculative* in nature: their failure is a regular event. Using backtracking for undoing the effect of a failed speculative computation is impossible in a concurrent context. Most computations including interoperating with the external world cannot backtrack. The essential idea to deal with speculative computations in a concurrent context is to *encapsulate* speculative computation so that the failure of an encapsulated computation has no effect on other computations.

Computation spaces. The idea of encapsulation has been pioneered by AKL, where encapsulation has been achieved by delegating computations to so-called deep guards (to be discussed later in more detail). Oz generalizes this idea as follows. Computations (roughly consisting of threads of statements and a constraint store) are contained in a *computation space*. Encapsulation in Oz then is achieved by delegating speculative computations to *local* computation spaces. The failure of a local space leaves other spaces unaffected.

Computation spaces can then be nested freely resulting in a tree of nested computation spaces as sketched in Figure 1.11. Encapsulation prevents that constraints told by computations in local computation spaces are visible in spaces higher up in the space tree. Nested propagation makes sure that constraints told in computation spaces are propagated to nested spaces.

`NewSpace`	:	*Script* $\rightarrow$ *Space*
`Ask`	:	*Space* $\rightarrow$ *Status*
`Access`	:	*Space* $\rightarrow$ *Solution*
`Clone`	:	*Space* $\rightarrow$ *Space*
`Commit`	:	*Space* $\times$ *Int* $\rightarrow$ *Unit*
`Inject`	:	*Space* $\times$ *Script* $\rightarrow$ *Unit*

Figure 1.12: Operations on first-class computation spaces.

Stability. Given a setup with local spaces for encapsulation, it is essential to have a criteria when a computation is not any longer speculative. A ground-breaking idea introduced by Janson and Haridi in the context of AKL is *stability* [60, 47, 59]. A speculative computation becomes *stable*, if it has entirely reduced to constraints and that these constraints are entailed or disentailed (that is, the constraints do not make any speculative assumptions themselves) by the constraints from computation spaces higher up in the space tree.

Stability naturally generalizes the notion of entailment by capturing when arbitrary computations are not any longer speculative. In particular, both entailment and stability are monotonic conditions: a stable computation space remains stable regardless of other computations.

Deep guards. Stability has been first used as a control criteria for combinators using so-called *deep guards*. A combinator can be disjunction, negation, or conditional, for example. In the concurrent constraint programming model guards (that is, ask statements) are *flat* as they are restricted to constraints. Deep guards allow arbitrary statements (agents) of the programming language. Similar to how entailment defines when and how computation can proceed for a flat guard, stability defines when and how computation can proceed for a deep guard.

First-class computation spaces. Local computation spaces together with stability as control regime serve as the foundation for both search and combinators in Oz. A general idea in Oz is that important abstractions such as procedures, classes, and objects are available as first-class citizens in the language. As discussed in Section 1.2.4, this is achieved by names that separate reference to entities from the entities proper.

Similarly, local computation spaces are available as first-class computation spaces. Having spaces available first-class, search and combinators become programmable within Oz as programming language.

The operations on first-class computation spaces are listed in Figure 1.12. `NewSpace` takes a script (a procedure that defines the constraint problem to be solved) and returns a space that executes the script. `Ask` synchronizes until computation in the space has reached a stable state. It then returns the status of the space, that is, whether the space is `failed`, `solved`, or has `alternatives`. Alternatives are then resolved by search. `Access` returns the solution stored in a space. `Clone` returns a copy of a space. `Commit` selects an alternative of a choice point. `Inject` adds constraints to a space. How the operations are employed for programming search becomes is sketched briefly below.

```
fun {All S}
   case {Ask S}
   of failed then nil
   [] solved then [{Access S}]
   [] alternatives then C={Clone S} in
      {Commit S 1} {Commit C 2}
      {Append {All S} {All C}}
   end
end
```

Figure 1.13: Depth-first exploration for all solutions.

Programming search. Most constraint programming systems (see Chapter 14, "Finite Domain Constraint Programming Systems") have in common that they offer a fixed and small set of search strategies. The strategies covered are typically limited to single, all, and best-solution search. Search cannot be programmed, which prevents users to construct new search strategies. Search hard-wires depth-first exploration, which prevents even system developers to construct new search strategies. With first-class computation spaces, Oz provides a mechanism to easily program arbitrary search engines featuring arbitrary exploration strategies.

Figure 1.13 conveys that programming search based on first-class computation spaces is easy. The figure contains a formulation of depth-first exploration that returns all solutions. `All` takes a space `S` containing the problem to be solved as input. It returns either the empty list, if no solution is found, or a singleton list containing the solution. If a space needs to be resolved by search, the space is copied (by application of `Clone`) and exploration follows the left alternative (`Commit S 1`) and later the right alternative (`Commit C 2`). `Append` then appends the solutions obtained from exploring both `S` and `C`.

The complete search engine is obtained by adding space creation according to the problem `P` (specified by a procedure `P`) to be solved:

```
fun {SearchAll P}
   {All {NewSpace P}}
end
```

First-class computation spaces not only cover many standard search engines but have been applied to interactive visual search [93], parallel search [91], and recomputation-based search [94]. A complete treatment of search with first-class computation spaces is [92]. Abstractions similar to first-class computation spaces are also used in the C++-based libraries Figaro [53] and Gecode [44].

Programming combinators. First-class computations spaces can also be used to program deep-guard combinators such as disjunction, negation, blocking implication, for example. Here the motivation is the same as for programming search: the user is not restricted to a fixed set of combinators but can devise application-specific combinators when needed. By this they generalize the idea of deep-guard combinators introduced in AKL. Programming combinators is covered in [90] and more extensively in [92].

1.3 Rule-Based Languages

Rule-based formalisms are ubiquitous in computer science, from theory to practice, from modelling to implementation, from inference rules and transition rules to business rules. Executable rules are used in declarative programming languages, in program transformation and analysis, and for reasoning in artificial intelligence applications. Rules consist of a data description (pattern) and a replacement statement for data matching that description. Rule applications cause localized transformations of a shared data structure (e.g., constraint store, term, graph, database). Applications are repeated until no more change happens.

Constraint Handling Rules (CHR) is a rule-based programming language in the tradition of constraint logic programming, the only one specifically developed for the implementation of constraint solvers. It is traditionally an extension to other programming languages but has been used increasingly as a general-purpose programming language, because it can embed many rule-based formalisms and describe algorithms in a declarative way.

The next section discusses design objectives and related work. Then we give an overview of syntax and semantics of CHR [35, 42] as well as of properties for program analysis such as confluence and operational equivalence. Then we give constraint solvers written in CHR, for Booleans, minima, arithmetic equations, finite and interval domains and lexicographic orders.

1.3.1 Design Objectives

Constraint solver programming. In the beginning of CLP, constraint solving was hardwired in a built-in constraint solver written in a low-level procedural language. While efficient, this so-called *black-box* approach makes it hard to modify a solver or build a solver over a new domain, let alone debug, reason about and analyse it. Several proposals have been made to allow more for flexibility and costumization of constraint solvers (called *glass-box*, sometimes *white-box* or even *no-box* approaches):

- Demons, forward rules and conditionals of the CLP language CHIP [29], allow defining propagation of constraints in limited ways.
- Indexicals, clp(FD) [25], allow implementing constraints over finite domains at a medium level of abstraction.
- Given constraints connected to a Boolean variable that represents their truth [16, 97] allow expressing any logical formula over primitive constraints.
- Constraint combinators, cc(FD) [109], allow building more complex constraints from simpler constraints.

All approaches extend a solver over a given, specific constraint domain, typically finite domains. The goal then was to design a programming language specifically for writing constraint solvers. Constraint Handling Rules (CHR) [35, 42, 11, 86] is a concurrent committed-choice constraint logic programming language consisting of guarded rules that transform multi-sets of relations called constraints until no more change happens.

Underlying concepts. CHR was motivated by the inference rules that are traditionally used in computer science to define logical relationships and fixpoint computation in the most abstract way.

In CHR, one distinguishes two main kinds of rules: *Simplification rules* replace constraints by simpler constraints while preserving logical equivalence, e.g., $X \leq Y \wedge Y \leq X \Leftrightarrow X = Y$. *Propagation rules* add new constraints that are logically redundant but may cause further simplification, e.g., $X \leq Y \wedge Y \leq Z \Rightarrow X \leq Z$. Obviously, conjunctions in the head of a rule and propagation rules are essential in expressing constraint solving succinctly.

Given a logical calculus and its transformation rules for deduction, its (conditional) inference rules directly map to propagation rules and its (biconditional) replacement rules to simplification rules. Also, the objects of logic, the (constraint) theories, are usually specified by implications or logical equivalences, corresponding to propagation and simplficiation rules.

Given a state transition system, its transition rules can readily be expressed with simplification rules. In this way, dynamics and changes (e.g., updates) can be modelled, possibly triggered by events and handled by actions. This justifies the use of CHR as a general purpose programming language.

Design influences. The design of CHR has many roots and combines their attractive features in a novel way. Logic programming (LP), constraint logic programming (CLP) [65, 42] and concurrent committed-choice logic programming (CCP) [95, 80] are direct ancestors of CHR. Like automated theorem proving, CHR uses formulae to derive new information, but only in a restricted syntax (e.g., no negation) and in a directional way (e.g., no contrapositives) that makes the difference between the art of proof search and an efficient programming language.

CHR adapts concepts from term rewriting systems [14] for program analysis, but goes beyond term rewriting by working on conjunctions of relations instead of nested terms, and by providing in the language design propagation rules, logical variables, built-in constraints, implicit constraint stores, and more. Extensions of rewriting, such as rewriting Logic [68] and its implementation in Maude [24] and Elan [19] have similar limitations as standard rewriting systems for writing constraints. The functional language Bertrand [63] uses augmented term rewriting to implement constraint-based languages.

Executable rules with multiple head atoms were proposed in the literature to model parallelism and distributed agent processing as well as objects [15, 12], but not for constraint solving. Other influences for the design of CHR were the Gamma computation model and the chemical abstract machine [15], and, of course, production rule systems like OPS5 [20].

Independent developments related to the concepts behind CHR were the multi-paradigm programming languages CLAIRE [22], and Oz [98] as well as database research: constraint and deductive databases, integrity constraints, and event-condition-action rules.

Expressiveness. The paper [101] introduces CHR machines, analogous to RAM and Turing machines. It shows that these machines can simulate each other in polynomial time, thus establishing that CHR is Turing-complete and, more importantly, that every algorithm can be implemented in CHR with best known time and space complexity, something that is not known to be possible in other pure declarative programming languages like Prolog.

Applications. Recent CHR libraries exist for most Prolog systems, e.g., [54, 84], Java [10, 118, 117, 66], Haskell [23] and Curry. Standard constraint systems as well as novel ones such as temporal, spatial or description logic constraints have been implemented in CHR. Over time CHR has proven useful outside its original field of application in constraint reasoning and computational logic[2], be it agent programming, multi-set rewriting or production rule systems: Recent applications of CHR range from type systems [31] and time tabling [5] to ray tracing and cancer diagnosis [11, 86]. In some of these applications, conjunctions of constraints are best regarded as interacting collections of concurrent agents or processes. We will not discuss CHR as a general-purpose programming language for space reasons.

Abstract Syntax

We distinguish between two different kinds of constraints: *built-in (pre-defined) constraints* which are solved by a built-in constraint solver, and CHR *(user-defined) constraints* which are defined by the rules in a CHR program. Built-in constraints include syntactic equality $=$, *true*, and *false*. This distinction allows to embed and utilize existing constraint solvers as well as side-effect-free host language statements. Built-in constraint solvers are considered as black-boxes in whose behavior is trusted and that do not need to be modified or inspected. The solvers for the built-in constraints can be written in CHR itself, giving rise to a hierarchy of solvers [87].

A CHR *program* is a finite set of rules. There are three kinds of rules:

Simplification rule:	*Name* $@\ H \Leftrightarrow C \mid B$
Propagation rule:	*Name* $@\ H \Rightarrow C \mid B$
Simpagation rule:	*Name* $@\ H \setminus H' \Leftrightarrow C \mid B$

Name is an optional, unique identifier of a rule, the *head* H, H' is a non-empty conjunction of CHR constraints, the *guard* C is a conjunction of built-in constraints, and the *body* B is a goal. A *goal* is a conjunction of built-in and CHR constraints. A trivial guard expression "*true*$\mid$" can be omitted from a rule.

Simpagation rules abbreviate simplification rules of the form $H \wedge H' \Leftrightarrow C \mid H \wedge B$, so there is no further need to discuss them separately.

Operational Semantics

At runtime, a CHR program is provided with an initial state and will be executed until either no more rules are applicable or a contradiction occurs.

The operational semantics of CHR is given by a transition system (Fig. 1.14). Let P be a CHR program. We define the transition relation $\mapsto$ by two computation steps (transitions), one for each kind of CHR rule. *States* are goals, i.e., conjunctions of built-in and CHR constraints. States are also called *(constraint) stores*. In the figure, all upper case letters are meta-variables that stand for conjunctions of constraints. The constraint theory CT defines the semantics of the built-in constraints. G_{bi} denotes the built-in constraints of G.

[2] Integrating deduction and abduction, bottom-up and top-down execution, forward and backward chaining, tabulation and integrity constraints.

Simplify

If	$(r@H \Leftrightarrow C \mid B)$ is a fresh variant with variables $\bar{x}$ of a rule named r in P
and	$CT \models \forall\,(G_{bi} \rightarrow \exists \bar{x}(H{=}H' \wedge C))$
then	$(H' \wedge G) \mapsto_r (B \wedge G \wedge H{=}H' \wedge C)$

Propagate

If	$(r@H \Rightarrow C \mid B)$ is a fresh variant with variables $\bar{x}$ of a rule named r in P
and	$CT \models \forall\,(G_{bi} \rightarrow \exists \bar{x}(H{=}H' \wedge C))$
then	$(H' \wedge G) \mapsto_r (H' \wedge B \wedge G \wedge H{=}H' \wedge C)$

Figure 1.14: Computation steps of Constraint Handling Rules

Starting from an arbitrary initial goal, CHR rules are applied exhaustively, until a fixpoint is reached. A simplification rule $H \Leftrightarrow C \mid B$ *replaces* instances of the CHR constraints H by B provided the guard C holds. A propagation rule $H \Rightarrow C \mid B$ instead *adds* B to H. If new constraints arrive, rule applications are restarted. Computation stops in a failed final state if the built-in constraints become inconsistent. Trivial non-termination of the **Propagate** computation step is avoided by applying a propagation rule at most once to the same constraints (see the more concrete semantics in [1]).

In more detail, a rule is *applicable*, if its head constraints are matched by constraints in the current goal one-by-one and if, under this matching, the guard of the rule is implied by the built-in constraints in the goal. Any of the applicable rules can be applied, and the application cannot be undone, it is committed-choice.

A *computation (derivation)* of a goal G is a sequence $S_0, S_1, \ldots$ of states with $S_i \mapsto S_{i+1}$ beginning with the *initial state (query, problem)* $S_0 = G$ and ending in a final state or not terminating. A *final state (answer, solution)* is one where either no computation step is possible anymore or where the built-in constraints are inconsistent.

Example 1. *We define a* CHR *constraint for a partial order relation* $\leq$*:*

```
reflexivity    @ X≤X ⇔ true
antisymmetry   @ X≤Y ∧ Y≤X   ⇔ X=Y
transitivity   @ X≤Y ∧ Y≤Z   ⇒ X≤Z
```

The CHR *program implements reflexivity, antisymmetry, transitivity and redundancy in a straightforward way.*

Operationally the rule `reflexivity` *removes occurrences of constraints that match* $X \leq X$*. The rule* `antisymmetry` *means that if we find* $X \leq Y$ *as well as* $Y \leq X$ *in the current goal, we can replace them by the logically equivalent* $X{=}Y$*. The rule* `transitivity` *propagates constraints. It adds the logical consequence* $X \leq Z$ *as a redundant constraint, but does not remove any constraints.*

A computation of the goal $A \leq B \wedge C \leq A \wedge B \leq C$ *proceeds as follows (rules are applied to underlined constraints):*

$$\begin{array}{ll}
\underline{A \leq B} \wedge \underline{C \leq A} \wedge B \leq C & \mapsto_{\texttt{transitivity}} \\
A \leq B \wedge C \leq A \wedge \underline{B \leq C} \wedge \underline{C \leq B} & \mapsto_{\texttt{antisymmetry}} \\
\underline{A \leq B} \wedge \underline{C \leq A} \wedge B{=}C & \mapsto_{\texttt{antisymmetry}} \\
A{=}B \wedge B{=}C &
\end{array}$$

Starting from a circular relationship, we have found out that the three variables must be the same.

Refined, parallel and compositional semantics. The high-level description of the operational semantics of CHR given here does not explicitly address termination at failure and of propagation rules, and leaves two main sources of non-determinism: the order in which constraints of a query are processed and the order in which rules are applied (rule scheduling). As in Prolog, almost all CHR implementations execute queries from left to right and apply rules top-down in the textual order of the program. This behavior has been formalized in the so-called *refined semantics* [32] that was proven to be a concretization of the standard operational semantics given in [1]. In [41] a *parallel* execution model for CHR is presented.

Search. Search in CHR is usally provided by the host language, e.g., by the built-in backtracking of Prolog or by search libraries in Java. In addition, in all Prolog implementations of CHR, the disjunction of Prolog can be used in the body of CHR rules. This was formalized in the language CHR$^\vee$ [7, 8]. An early implementation of CHR in Eclipse Prolog also featured so-called labeling declarations [35], that allowed Prolog clauses for CHR constraints. These can be directly translated into CHR$^\vee$, which we will use to define labeling procedures.

Pragmatics. When writing CHR programs, manuals such as [54] suggest to prefer simplification rules and to avoid propagation rules and multiple heads (although indexing often helps to find partner constraints in constant time [84]). One will often modify and compose existing CHR and other programs. Some possibilities are: Flat composition by taking the union of all rules [4]; hierarchical composition by turning some CHR constraints into built-in constraints of another constraint solver [89]; extending arbitrary solvers with CHR [30]. CHR are usually combined with a host language. In the host language, CHR constraints can be posted; in the CHR rules, host language statements can be included as built-in constraints.

Declarative Semantics

Owing to the tradition of logic and constraint logic programming, CHR features – besides an operational semantics – a *declarative semantics*, i.e., a direct translation of a CHR program into a first-order theory. In the case of constraint solvers, this strongly facilitates proofs of a program's faithful handling of constraints.

The *logical reading (meaning) of simplification and propagation rules* is given below.

$$H \Leftrightarrow C \mid B \quad \forall(C \rightarrow (H \leftrightarrow \exists \bar{y}\ B))$$
$$H \Rightarrow C \mid B \quad \forall(C \rightarrow (H \rightarrow \exists \bar{y}\ B))$$

The sequence $\bar{y}$ are the variables that appear only in the body B of a rule.

The *logical reading of a* CHR *program* is the conjunction of the logical readings of its rules united with the constraint theory CT that defines the built-in constraints. The *logical reading of a state* is just the conjunction of its constraints. State transitions preserve logical equivalence, i.e., all states in a computation are logically the same. From this result,

soundness and completeness theorems follow that show that the declarative and operational semantics coincide strongly, in particular if the program is confluent [9].

Linear-logic semantics. The classical-logic declarative semantics, however, does not suffice when CHR is used as a general-purpose concurrent programming language. Many algorithms do not have a correct first-order logic reading, especially when they crucially rely on change through updates. This problem has been demonstrated in [41, 85] and led to the development of an alternative declarative semantics. It is based on a subset of *linear logic* [45] that can model resource consumption. It therefore more accurately describes the operational behavior of simplification rules [18].

Program Properties and Their Analysis

One advantage of a declarative programming language is the ease of program analysis. The paper [27] introduces a fix-point semantics which characterizes the input/output behavior of a CHR program and which is *and*-compositional. It allows to retrieve the semantics of a conjunctive query from the semantics of its conjuncts. Such a semantics can be used as a basis to define incremental and modular program analysis and verification tools. An abstract interpretation framework for CHR is introduced in [88]. The basic properties of termination, confluence and operational equivalence are traditionally analysed using specific techniques as discussed below. Time complexity analysis is discussed in [36], but details often rely on problem specific techniques.

Minimal states. When analysing properties of CHR programs that involve the infinitely many possible states, we can sometimes restrict ourselves to a finite number of so-called minimal states. For each rule, there is a minimal, most general state to which it is applicable. This state is the conjunction of the head and the guard of the rule. Removing any constraint from the state would make the rule inapplicable. Every other state to which the rule is applicable contains the minimal state. Adding constraints to the state cannot inhibit the applicability of a rule because of the *monotonicity property* of CHR [9].

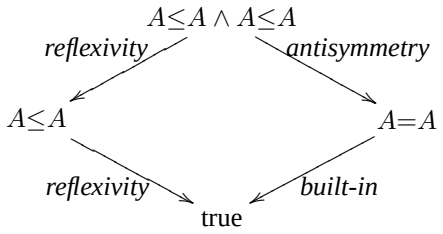
Termination. A CHR program is called *terminating*, if there are no infinite computations. Since CHR is Turing-complete, termination is undecidable. For CHR programs that mainly use simplification rules, simple well-founded orderings are often sufficient to prove termination [37, 36]. For CHR programs that mainly use propagation rules, results from bottom-up logic programming [43] as well as deductive and constraint databases apply. In general, termination analysis is difficult for non-trivial interactions between simplification and propagation rules.

Confluence. In a CHR program, the result of computations from a given goal will always have the same meaning. However the answer may not be syntactically the same. The confluence property of a program guarantees that any computation for a goal results in the same final state no matter which of the applicable rules are applied.

The papers [1, 9] give a decidable, sufficient and necessary condition for confluence: A terminating CHR program is confluent if and only if all its critical pairs are joinable. For checking confluence, one takes two rules (not necessarily different) from the program.

The minimal states of the rules are overlapped by equating at least one head constraint from one rule with one from the other rule. For each *overlap*, we consider the two states resulting from applying one or the other rule. These two states form a so-called *critical pair*. One tries to *join* the states in the critical pair by finding two computations starting from the states that reach a common state. If the critical pair is not joinable, we have found a counterexample for confluence of the program.

Example 2. *Recall the program for $\leq$ of Example 1. Consider the rules for reflexivity and antisymmetry and overlap them to get the following critical state and computations.*



The resulting critical pair is obviously joinable. The example also shows that multiplicities matter in CHR.

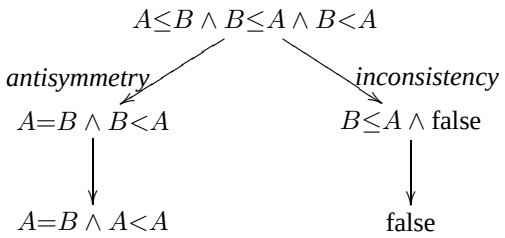
Any terminating and confluent CHR program has a consistent logical reading [9, 1] and will automatically implement a concurrent any-time (approximation) and on-line (incremental) algorithm.

Completion. Completion is the process of adding rules to a non-confluent program until it becomes confluent. Rules are built from a non-joinable critical pair to allow a transition from one of the states into the other while maintaining termination.

Example 3. *Given the $\leq$ solver, assume we want to introduce a $<$ constraint by adding just one rule about the interaction between these two types of inequalities.*

$$X{\leq}Y \wedge Y{\leq}X \quad \Leftrightarrow \quad X{=}Y \qquad \textit{(antisymmetry)}$$
$$X{\leq}Y \wedge Y{<}X \quad \Leftrightarrow \quad \text{false} \qquad \textit{(inconsistency)}$$

The resulting program is not confluent.



Completion uses the two non-joinable states to derive an interesting new rule, discovering irreflexivity of $<$.

$$X{<}X \Leftrightarrow \text{false}$$

In contrast to other completion methods, in CHR we generally need more than one rule to make a critical pair joinable: a simplification rule and a propagation rule [3].

Operational equivalence. A fundamental and hard question in programming language semantics is when two programs should be considered equivalent. For example correctness of program transformation can be studied only with respect to a notion of equivalence. Also, if modules or libraries with similar functionality are used together, one may be interested in finding out if program parts in different modules or libraries are equivalent. In the context of CHR, this case arises frequently when constraint solvers written in CHR are combined. Typically, a constraint is only partially defined in a constraint solver. We want to make sure that the operational semantics of the common constraints of two programs do not differ.

Two programs are operationally equivalent if for each goal, all final states in one program are the same as the final states in the other program. In [2], the authors gave a decidable, sufficient and necessary syntactic condition for operational equivalence of terminating and confluent CHR programs[3]: The minimal states of all rules in both programs are simply run as goals in both programs, and they must reach a common state. An example for operational equivalence checking can be found with the minimum example in Section 1.3.2.

1.3.2 Constraint Solvers

We introduce some constraint solvers written in CHR, for details and more solvers see [38, 42]. We will use the concrete ASCII syntax of CHR implementations in Prolog: Conjunction $\wedge$ is written as comma ','. Disjunction $\vee$ is written as semi-colon ';'. Let '=<' and '<' be built-in constraints now.

Boolean Constraint Solver

Boolean algebra (propositional logic) constraints can be solved by different techniques [67]. The logical connectives are represented as Boolean constraints, i.e., in relational form. For example, conjunction is written as the constraint `and(X,Y,Z)`, where `Z` is the result of anding `X` and `Y`. In the following terminating and confluent Boolean constraint solver [42], a local consistency algorithm is used. It simplifies one Boolean constraint at a time into one or more syntactic equalities whenever possible. The rules for propositional conjunction are as follows.

```
and(X,Y,Z) <=> X=0 | Z=0.
and(X,Y,Z) <=> Y=0 | Z=0.
and(X,Y,Z) <=> X=1 | Y=Z.
and(X,Y,Z) <=> Y=1 | X=Z.
and(X,Y,Z) <=> X=Y | Y=Z.
and(X,Y,Z) <=> Z=1 | X=1,Y=1.
```

The above rules are based on the idea that, given a value for one of the variables in a constraint, we try to determine values for other variables. However, the Boolean solver goes beyond propagating values, since it also propagates equalities between variables. For example, `and(1,Y,Z),neg(Y,Z)` will reduce to `false`, and this cannot be achieved by value propagation alone.

[3] To the best of our knowledge, CHR is the only programming language in practical use that admits decidable operational equivalence.

Search. The above solver is incomplete. For example, the solver cannot detect inconsistency of `and(X,Y,Z),and(X,Y,W),neg(Z,W)`. For completeness, constraint solving has to be interleaved with search. For Boolean constraints, search can be done by trying the values `0` or `1` for a variable. The generic labeling procedure `enum` traverses a list of variables.

```
enum([]) <=> true.
enum([X|L]) <=> indomain(X), enum(L).

indomain(X) <=> (X=0 ; X=1).
```

Minimum Constraint

The CHR constraint `min(X,Y,Z)` means that `Z` is the minimum of `X` and `Y`.

```
r1 @ min(X,Y,Z) <=> X=<Y | Z=X.
r2 @ min(X,Y,Z) <=> Y=<X | Z=Y.
r3 @ min(X,Y,Z) <=> Z<X  | Y=Z.
r4 @ min(X,Y,Z) <=> Z<Y  | X=Z.
r5 @ min(X,Y,Z) ==> Z=<X, Z=<Y.
```

The first two rules `r1` and `r2` correspond to the usual definition of `min`. But we also want to be able to compute backwards. So the two rules `r3` and `r4` simplify `min` if the order between the result `Z` and one of the input variables is known. The last rule `r5` ensures that `min(X,Y,Z)` unconditionally implies `Z=<X,Z=<Y`. Rules such as these can be automatically generated from logical specifications [6].

Example 4. *Redundancy from a propagation rule is useful, as the goal `min(A,2,2)` shows. To this goal only the propagation rule is applicable, but to the resulting state the second rule becomes applicable:*

$$\begin{array}{ll} & \texttt{min(A,2,2)} \\ \mapsto_{r5} & \texttt{min(A,2,2)}, \texttt{2=<A} \\ \mapsto_{r2} & \texttt{2=<A} \end{array}$$

In this way, we find out that for `min(A,2,2)` to hold, `2=<A` must hold.
Another interesting derivation involving the propagation rule is:

$$\begin{array}{ll} & \texttt{min(A,B,M)}, \texttt{A=<M} \\ \mapsto_{r5} & \texttt{min(A,B,M)}, \texttt{A=M}, \texttt{M=<B} \\ \mapsto_{r1} & \texttt{A=M}, \texttt{M=<B} \end{array}$$

It can be shown that the program is terminating and confluent. For example, the only overlap of the minimal states for the first two rules, `r1` and `r2` is `min(X,Y,Z),X=Y`. For both rules, their application leads to logically equivalent built-in constraints `X=Y,Y=Z`.

Operational equivalence. We would like to know if these two CHR rules defining the user-defined constraint `min` with differing guards

```
min(X,Y,Z) <=> X=<Y | Z=X.
min(X,Y,Z) <=> Y<X  | Z=Y.
```

are operationally equivalent with these two rules

```
min(X,Y,Z) <=> X<Y  | Z=X.
min(X,Y,Z) <=> Y=<X | Z=Y.
```

or if the union of the rules results in a better constraint solver for `min`.

Already the minimal state of the first rule of the first program, `min(X,Y,Z),X=<Y`, shows that the two programs are not operationally equivalent, since it can reduce to `Z=X` in the first program, but is a final state for the second program, since `X=<Y` does not apply any of the guards in the second program. Thus the union of the two programs allows for more constraint simplification. In the union, the two rules with the strict guards can be removed as another operational equivalence test shows that they are redundant.

Linear Polynomial Equation Solving

Typically, in arithmetic constraint solvers, incremental variants of classical variable elimination algorithms [58] like Gaussian elimination for equations and Dantzig's Simplex algorithm for equations are implemented.

A conjunction of equations is *in solved form* if the left-most variable of each equation does not appear in any other equation. We compute the solved form by eliminating multiple occurrences of variables. In this solved form, all determined variables (those that take a unique value) are discovered.

```
eliminate @ A1*X+P1 eq 0 \ P2X eq 0 <=>
                           find(A2*X,P2X,P2) |
                           normalize(A2*(-P1/A1)+P2,P3),
                           P3 eq 0.

constant  @ B eq 0 <=> number(B) | zero(B).
```

The `constant` rule says that if the polynomial contains no more variables, then the number `B` must be zero. The `eliminate` rule performs variable elimination. It takes any pair of equations with a common occurrence of a variable, `X`. In the first equation, the variable appears left-most. This equation is used to eliminate the occurrence of the variable in the second equation. The first equation is left unchanged. In the guard, the built-in `find(A2*X,P2X,P2)` tries to find the expression `A2*X` in the polynom `P2X`, where `X` is the common variable. The polynom `P2` is `P2X` with `A2*X` removed. The constraint `normalize(E,P)` transforms an arithmetic expression `E` into a linear polynomial `P`.

The solver is complete, so no search is necessary. It is terminating but not confluent due to the `eliminate` rule: Consider two equations with the same left-most variable, then the rule can be applied in two different ways. The solver produces the solved form as can be shown by contradiction: If a set of equations is not in solved form, then the `eliminate` rule is applicable. The solver is concurrent by nature of CHR: It can reduce pairs of equations in parallel or eliminate the occurence of a variable in all other equations at once.

Finite Domains

Here, variables are constrained to take their value from a given, finite set. Choosing integers for values allows for arithmetic expressions as constraints. Influential CLP languages with finite domains are CHIP [29], clp(FD) [25] and cc(FD) [109].

The *domain constraint* $X \; in \; D$ means that the variable X takes its value from the given finite domain D. For simplicity, we start with the *bounds consistency* algorithm for interval constraints [108, 17]. The implementation is based on interval arithmetic. In the solver, `in`, `le`, `eq`, `ne`, and `add` are CHR constraints, the inequalities `<`, `>`, `=<`, `>=`, and `<>` are built-in arithmetic constraints, and `min`, `max`, `+`, and `-` are built-in arithmetic functions. `X in A..B` constrains `X` to be in the interval `A..B`. The rules for local consistency affect the interval constraints (`in`) only, the other constraints remain unaffected.

```
inconsistency @ X in A..B <=> A>B | false.
intersect@ X in A..B, X in C..D <=> X in max(A,C)..min(B,D).

le @ X le Y, X in A..B, Y in C..D <=> B>D |
     X le Y, X in A..D, Y in C..D.
le @ X le Y, X in A..B, Y in C..D <=> C<A |
     X le Y, X in A..B, Y in A..D.

eq @ X eq Y, X in A..B, Y in C..D <=> A<>C |
     X eq Y, X in max(A,C)..B, Y in max(C,A)..D.
eq @ X eq Y, X in A..B, Y in C..D <=> B<>D |
     X eq Y, X in A..min(B,D), Y in C..min(D,B).
```

The CHR constraint `X le Y` means that `X` is less than or equal to `Y`. Hence, `X` cannot be larger than the upper bound `D` of `Y`. Therefore, if the upper bound `B` of `X` is larger than `D`, we can replace `B` by `D` without removing any solutions. Analogously, one can reason on the lower bounds to tighten the interval for `Y`. The `eq` constraint causes the intersection of the interval domains of its variables provided the bounds are not yet the same.

Example 5. *Here is a sample computation involving* `le`*:*

```
        U in 2..3, V in 1..2, U le V
↦le     V in 1..2, U le V, U in 2..2
↦le     U le V, U in 2..2, V in 2..2.
```

Finally, $X+Y=Z$ is represented as `add(X,Y,Z)`.

```
add @ add(X,Y,Z), X in A..B, Y in C..D, Z in E..F <=>
        not (A>=E-D,B=<F-C,C>=E-B,D=<F-A,E>=A+C,F=<B+D) |
      add(X,Y,Z),
      X in max(A,E-D)..min(B,F-C),
      Y in max(C,E-B)..min(D,F-A),
      Z in max(E,A+C)..min(F,B+D).
```

For addition, we use interval addition and subtraction to compute the interval of one variable from the intervals of the other two variables. The guard ensures that at least one interval becomes smaller whenever the rule is applied. Here is a sample computation involving `add`:

```
U in 1..3, V in 2..4, W in 0..4, add(U,V,W)  ↦add
add(U,V,W), U in 1..2, V in 2..3, W in 3..4
```

For termination, consider that the rules `inconsistency` and `intersection` from above remove one interval constraint each. We assume that the remaining rules deal with non-empty intervals only. This holds under the refined semantics and can be enforced by additional guard constraints on the interval bounds. Then in each rule, at least one interval in the body is strictly smaller than the corresponding interval in the head, while the other intervals remain unaffected. The solver is confluent, provided the intervals are given. The solver also works with intervals of real numbers of a choosen granularity, so that to ensure termination rules are not applied anymore to domains which are considered too small.

Enumeration domains. Besides intervals, finite domains can be explicit enumerations of possible values. The rules for enumeration domains are analogous to the ones for interval domains and implement arc consistency [73], for example:

```
inconsistency @ X in [] <=> false.
intersect@ X in L1,X in L2 <=> intersect(L1,L2,L3) | X in L3.
```

Search. We implement the search routine analogous to the one for Boolean constraints. For interval domains, search is usually done by splitting intervals in two halves. This splitting is repeated until the bounds of the interval are the same.

```
indomain(X), X in A..B <=> A<B |
            (X in A..(A+B)//2, indomain(X) ;
             X in (A+B)//2+1..B, indomain(X)).
```

The guard ensures termination. For enumeration domains, each value in the domain (implemented as a list) is tried. $X{=}V$ is expressed as `X in [V]`.

```
indomain(X), X in [V|L] <=> L=[_|_] |
            (X in [V] ; X in L, indomain(X)).
```

The guard ensures termination. Calling `indomain(X)` in the second disjunct ensures that subsequently, the next value for `X` from the list `L` will be tried.

N-queens. The famous n-queens problem asks to place n queens $q_1, \ldots, q_n$ on an $n * n$ chess board, such that they do not attack each other. The problem can be solved with a CHR program, where `N` is the size of the chess board and `Qs` is a list of `N` queen position variables.

```
solve(N,Qs) <=> makedomains(N,Qs), queens(Qs), enum(Qs).
queens([Q|Qs]) <=> safe(Q,Qs,1), queens(Qs).
safe(X,[Y|Qs],N) <=> noattack(X,Y,N), safe(X,Qs,N+1).
```

Instead of implementing `noattack` with the usual three finite domain inequality constraints, we can use `noattack` directly:

```
noattack(X,Y,N), X in [V], Y in D <=>
           remove(D,[V,V+N,V-N],D1) | Y in D1.
noattack(Y,X,N), X in [V], Y in D <=>
           remove(D,[V,V+N,V-N],D1) | Y in D1.
```

The constraint between three lists `remove(D,L,D1)` holds if `D1` is `D` without the values in `L` and at least one value has been removed.

Lexicographic Order Global Constraint

A lexicographic order $\preceq_{lex}$ (`lex`) allows to compare sequences by comparing the elements of the sequences proceeding from start to end. Given two sequences l_1 and l_2 of variables of the same length n, $[x_1,\ldots,x_n]$ and $[y_1,\ldots,y_n]$, then $l_1 \preceq_{lex} l_2$ if and only if $n{=}0$ or $x_1{<}y_1$ or $x_1{=}y_1$ and $[x_2,\ldots,x_n] \preceq_{lex} [y_2,\ldots,y_n]$.

The solver [40] consists of three pairs of rules, the first two corresponding to base cases of the recursion (garbage collection), then two rules performing forward reasoning (recursive traversal and implied inequality), and finally two for backward reasoning, covering a not so obvious special case when the lexicographic constraint has a unique solution.

```
l1 @ [] lex [] <=> true.
l2 @ [X|L1] lex [Y|L2] <=> X<Y | true.
l3 @ [X|L1] lex [Y|L2] <=> X=Y | L1 lex L2.
l4 @ [X|L1] lex [Y|L2] ==> X=<Y.

l5 @ [X,U|L1] lex [Y,V|L2] <=> U>V | X<Y.
l6 @ [X,U|L1] lex [Y,V|L2] <=> U>=V, L1=[_|_] |
              [X,U] lex [Y,V], [X|L1] lex [Y|L2].
```

The implementation is short and concise without giving up on linear time worst case time complexity. It is incremental and concurrent by nature of CHR. It is provably correct and confluent. It is independent of the underlying constraint system. In [40], also completeness of constraint propagation is shown, i.e., given a `lex` constraint and an inequality, all implied inequalities are generated by the solver.

1.4 Challenges and Opportunities

The integration of constraint technology in more traditional or hybrid paradigms has been a source of significant progress. Nonetheless, it is still shy of a comprehensive solution that addresses all the motivating objectives. It has, however, created flexible platforms particularly well-suited for experimenting with novel research ideas and directions. This section considers some of these opportunities.

1.4.1 Cooperative Solvers

Cooperative solvers are already a reality. Linear Programming and Integer Programming solvers have been used in conjunction with constraint solvers and the combination often proved quite effective. New solvers are developed regularly either for domain specific needs or as vertical extensions. In all cases, hybridization raises many issues: How should solvers communicate? How do solvers compose? What is the composite's architecture (side-by-side, master-slave, concurrent,...)? What are the synchronization triggers and events (variable bounds, heuristic information, objective function, impacts,...)? Should the solvers operate on redundant statements of the same problems or on disjoint subset of

constraints they are better suited for? Can solver-specific formulations be derived from a unique master statement? Can the formulations be automatically refined over time?

1.4.2 Orthogonal Computation Models

Recent developments in Constraint-Based Local Search [106] clearly indicate that constraint-based solvers can be developed for radically different computation models. From a declarative standpoint, local search solvers rely on constraints to specify the properties of solutions and write elegant, high-level, and reusable search procedures which automatically exploit the constraints to guide the search. From a computational standpoint, the solver incrementally maintains properties (e.g. truth value, violation degree, variable and value based violations) under non-monotonic changes to the decision variables that always have a tentative value assignment. This organization is a fundamental departure from classic domain-based consistency and filtering techniques found in traditional finite domain solvers.

The fundamental differences are related to the nature of the underlying computational models. How can these solvers be effectively hybridized? What steps are required for an efficient integration of the computation models that does not result in severe performance degradation for either? Once the two technologies coexist, how can the solvers be composed? How can each solver benefit from results produced by its counter-part? Which form of collaboration is most effective?

1.4.3 Orthogonal Concerns

As solvers sophistication increases, it becomes difficult to anticipate the behavior of a solver on a given problem formulation. The advances in solver technology (efficiency, flexibility, openness) should be matched with equal progress in supporting abstractions for model designers. For Rapid application development, it is essential to assist the developers of optimization models. Improvements should include better debugging tools (where debugging occurs at the abstraction level of the model), explanation tools for post-mortem analysis, but also tracing tools for live analysis of the solver's behavior during the search process. Tools like the OZ Explorer [93] or the tree visualizer of OplStudio [113] provide initial insights into the dynamics of the search but fail to relate this behavior to modeling abstractions (constraints) and their interplay. Novel tools should also support the exploration of alternative model formulation and search heuristics to quickly identify successful strategies, a task which becomes increasingly burdensome given the large number of potential heuristics that ought to be considered.

1.5 Conclusion

Constraint solving and handling has moved from logic programming into more common programming paradigms and faced the challenges that it found there.

- Generalizing search from built-in backtracking of Prolog to flexible search routines as in OPL, OZ and SALSA.
- User friendliness by providing well-known metaphors resulting in modelling languages such as OPL and Comet.

- Integration into advanced multi-paradigm languages such as CLAIRE and Oz.
- The move from black-box solvers to glass-box solvers, that can be customized and analysed more easily, with constraint handling rules (CHR) at the extreme end of the spectrum.

These issues will remain a topic of research and development in constraint programming for the near future, but impressive first steps have been done.

Acknowledgments

Christian Schulte is partially funded by the Swedish Research Council (VR) under grant 621-2004-4953.

Bibliography

[1] S. Abdennadher. Operational semantics and confluence of constraint propagation rules. In *3rd International Conference on Principles and Practice of Constraint Programming*, LNCS 1330. Springer, 1997.

[2] S. Abdennadher and T. Frühwirth. Operational equivalence of constraint handling rules. In *Fifth International Conference on Principles and Practice of Constraint Programming, CP99*, LNCS 1713. Springer, 1999.

[3] S. Abdennadher and T. Frühwirth. On completion of constraint handling rules. In *4th International Conference on Principles and Practice of Constraint Programming, CP98*, LNCS 1520. Springer, 1998.

[4] S. Abdennadher and T. Frühwirth. Integration and optimization of rule-based constraint solvers. In M. Bruynooghe, editor, *Logic Based Program Synthesis and Transformation - LOPSTR 2003, Revised Selected Papers*, LNCS 3018. Springer, 2004.

[5] S. Abdennadher and M. Marte. University course timetabling using Constraint Handling Rules. *Journal of Applied Artificial Intelligence*, 14(4):311–326, 2000.

[6] S. Abdennadher and C. Rigotti. Automatic generation of chr constraint solvers. *Theory Pract. Log. Program.*, 5(4-5):403–418, 2005. ISSN 1471-0684. doi: http://dx.doi.org/10.1017/S1471068405002371.

[7] S. Abdennadher and H. Schütz. Model generation with existentially quantified variables and constraints. In *6th International Conference on Algebraic and Logic Programming*, LNCS 1298. Springer, 1997.

[8] S. Abdennadher and H. Schütz. CHR$^{\vee}$: A flexible query language. In *Flexible Query Answering Systems*, LNAI 1495. Springer, 1998.

[9] S. Abdennadher, T. Frühwirth, and H. Meuss. Confluence and semantics of constraint simplification rules. *Constraints Journal, Special Issue on the 2nd International Conference on Principles and Practice of Constraint Programming*, 4(2): 133–165, 1999.

[10] S. Abdennadher, E. Krämer, M. Saft, and M. Schmauss. Jack: A java constraint kit. In *Electronic Notes in Theoretical Computer Science*, volume 64, 2000.

[11] S. Abdennadher, T. Frühwirth, and C. Holzbaur. Editors, Special Issue on Constraint Handling Rules. *Theory and Practice of Logic Programming*

(TPLP), 5(4–5), 2005. URL http://www.informatik.uni-ulm.de/pm/mitarbeiter/fruehwirth/tplp-chr/ind%ex.html.

[12] J.-M. Andreoli and R. Pareschi. Linear objects: logical processes with built-in inheritance. In *7th International Conference on Logic programming (ICLP)*, pages 495–510, Cambridge, MA, USA, 1990. MIT Press. ISBN 0-262-73090-1.

[13] J. Armstrong, R. Virding, and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall International, Englewood Cliffs, NY, USA, 1993.

[14] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge Univ. Press, 1998.

[15] J.-P. Banatre, A. Coutant, and D. L. Metayer. A parallel machine for multiset transformation and its programming style. *Future Generation Computer Systems*, 4: 133–144, 1988.

[16] F. Benhamou. Interval constraint logic programming. In A. Podelski, editor, *Constraint Programming: Basics and Trends*, LNCS 910, pages 1–21. Springer, 1995.

[17] F. Benhamou and W. J. Older. Applying interval arithmetic to real, integer, and boolean constraints. *The Journal of Logic Programming*, 32(1), 1997.

[18] H. Betz and T. Frühwirth. A linear-logic semantics for constraint handling rules. In P. van Beek, editor, *11th Conference on Principles and Practice of Constraint Programming CP 2005*, volume 3709 of *Lecture Notes in Computer Science*, pages 137–151. Springer, Oct. 2005. URL http://www.informatik.uni-ulm.de/pm/mitarbeiter/fruehwirth/Papers/llchr%-final0.pdf.

[19] P. Borovansky, C. Kirchner, H. Kirchner, P. E. Moreau, and M. Vittek. ELAN: A logical framework based on computational systems. In *Proc. of the First Int. Workshop on Rewriting Logic*, volume ENTCS 4(1). Elsevier, 2004. URL citeseer.ist.psu.edu/borovansky97elan.html.

[20] L. Brownston, R. Farrell, E. Kant, and N. Martin. *Programming expert systems in OPS5: an introduction to rule-based programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1985. ISBN 0-201-10647-7.

[21] Y. Caseau, P.-Y. Guillo, and E. Levenez. A Deductive and Object-Oriented Approach to a Complex Scheduling Problem. In *Proc. of DOOD'93*, Phoenix, AZ, December 1989.

[22] Y. Caseau, F.-X. Josset, and F. Laburthe. Claire: combining sets, search and rules to better express algorithms. *Theory Pract. Log. Program.*, 2(6):769–805, 2002. ISSN 1471-0684. doi: http://dx.doi.org/10.1017/S1471068401001363.

[23] W.-N. Chin, M. Sulzmann, and M. Wang. A type-safe embedding of constraint handling rules into haskell. Technical report, School of Computing, National University of Singapore, Singapore, 2003.

[24] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: specification and programming in rewriting logic. *Theor. Comput. Sci.*, 285(2):187–243, 2002. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/S0304-3975(01)00359-0.

[25] P. Codognet and D. Diaz. Compiling constraints in clp(FD). *Journal of Logic Programming*, 27(3):185–226, 1996.

[26] *Mosel: An Overview*. Dash Optimization White Paper, 2004. http://www.dashoptimization.com/home/products/products_mosel.html.

[27] G. Delzanno, M. Gabbrielli, and M. C. Meo. A compositional semantics for chr. In *PPDP '05: Proceedings of the 7th ACM SIGPLAN international conference on*

Principles and practice of declarative programming, pages 209–217, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-090-6. doi: http://doi.acm.org/10.1145/1069774.1069794.

[28] D. Diaz and P. Codognet. A minimal extension of the `WAM` for `CLP(FD)`. In *Proceedings of the Tenth International Conference on Logic Programming (ICLP-93)*, pages 774–792, Budapest (Hungary), June 1993.

[29] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In *International Conference on Fifth Generation Computer Systems*, pages 693–702. Institute for New Generation Computer Technology, 1988.

[30] G. J. Duck, P. J. Stuckey, M. G. de la Banda, and C. Holzbaur. Extending arbitrary solvers with constraint handling rules. In *PPDP '03: Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming*, pages 79–90, New York, NY, USA, 2003. ACM Press. ISBN 1-58113-705-2. doi: http://doi.acm.org/10.1145/888251.888260.

[31] G. J. Duck, S. L. P. Jones, P. J. Stuckey, and M. Sulzmann. Sound and decidable type inference for functional dependencies. In *ESOP*, pages 49–63, 2004.

[32] G. J. Duck, P. J. Stuckey, M. G. de la Banda, and C. Holzbaur. The Refined Operational Semantics of Constraint Handling Rules. In B. Demoen and V. Lifschitz, editors, *20th International Conference on Logic Programming (ICLP)*, LNCS. Springer, 2004.

[33] R. Fourer, K. Martin, and J. Ma. Modeling systems & optimization services. Book in preparation.

[34] R. Fourer, D. Gay, and B. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. The Scientific Press, San Francisco, CA, 1993.

[35] T. Frühwirth. Theory and Practice of Constraint Handling Rules, Special Issue on Constraint Logic Programming. *Journal of Logic Programming*, 37(1–3):95–138, 1998. URL `http://www.pst.informatik.uni-muenchen.de/personen/fruehwir/drafts/jlp-%chr1.ps.Z`.

[36] T. Frühwirth. As Time Goes By: Automatic Complexity Analysis of Simplification Rules. In *8th International Conference on Principles of Knowledge Representation and Reasoning*, Toulouse, France, 2002.

[37] T. Frühwirth. Proving termination of constraint solver programs. In E. M. K.R. Apt, A.C. Kakas and F. Rossi, editors, *New Trends in Constraints*, LNAI 1865. Springer, 2000.

[38] T. Frühwirth. Constraint systems and solvers for constraint programming. *Special Issue of Archives of Control Sciences (ACS) on Constraint Programming for Decision and Control*, 2006. URL `http://www.informatik.uni-ulm.de/pm/mitarbeiter/fruehwirth/Papers/acs-s%ystems3.pdf`. To appear.

[39] T. Frühwirth. Constraint handling rules. In A. Podelski, editor, *Constraint Programming: Basics and Trends*, LNCS 910. Springer, March 1995.

[40] T. Frühwirth. Complete propagation rules for lexicographic order constraints over arbitrary domains. In *Recent Advances in Constraints, CSCLP 2005*, LNAI. Springer, 2006. To appear.

[41] T. Frühwirth. Parallelizing union-find in constraint handling rules using confluence. In M. Gabbrielli and G. G., editors, *Logic Programming: 21st International Con-*

ference, ICLP 2005, volume 3668 of *Lecture Notes in Computer Science*, pages 113–127. Springer, Oct. 2005. URL `http://www.informatik.uni-ulm.de/pm/mitarbeiter/fruehwirth/Papers/puf0.%pdf`.

[42] T. Frühwirth and S. Abdennadher. *Essentials of Constraint Programming*. Springer, 2003.

[43] H. Ganzinger and D. McAllester. A new meta-complexity theorem for bottom-up logic programs. In *International Joint Conference on Automated Reasoning*, LNCS 2083, pages 514–528. Springer, 2001.

[44] Gecode Team. Gecode (generic constraint development environment), 2005. Available from `www.gecode.org`.

[45] J.-Y. Girard. Linear logic: Its syntax and semantics. *Theoretical Computer Science*, 50:1–102, 1987.

[46] C. Guéret, C. Prins, M. Sevaux, and S. Heipcke. *Applications of Optimization with XpressMP*. Dash Optimization Ltd., 2002.

[47] S. Haridi, S. Janson, and C. Palamidessi. Structural operational semantics for AKL. *Future Generation Computer Systems*, 8:409–421, 1992.

[48] S. Haridi, P. Van Roy, P. Brand, and C. Schulte. Programming languages for distributed applications. *New Generation Computing*, 16(3):223–261, 1998.

[49] S. Haridi, P. Van Roy, P. Brand, M. Mehl, R. Scheidhauer, and G. Smolka. Efficient logic variables for distributed computing. *ACM Transactions on Programming Languages and Systems*, 21(3):569–626, May 1999.

[50] W. Harvey and M. Ginsberg. Limited Discrepancy Search. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, Montreal, Canada, August 1995.

[51] M. Henz. *Objects for Concurrent Constraint Programming*, volume 426 of *International Series in Engineering and Computer Science*. Kluwer Academic Publishers, Boston, MA, USA, Oct. 1997.

[52] M. Henz, G. Smolka, and J. Würtz. Oz—A programming language for multi-agent systems. In *13th International Joint Conference on Artificial Intelligence*, volume 1, pages 404–409, Chambéry, France, 1993. Morgan Kaufmann Publishers. Revised version appeared as [**?**].

[53] M. Henz, T. Müller, and K. B. Ng. Figaro: Yet another constraint programming library. In I. de Castro Dutra, V. S. Costa, G. Gupta, E. Pontelli, M. Carro, and P. Kacsuk, editors, *Parallelism and Implementation Technology for (Constraint) Logic Programming*, pages 86–96, Las Cruces, NM, USA, Dec. 1999. New Mexico State University.

[54] C. Holzbaur and T. Frühwirth. *Constraint Handling Rules Reference Manual for Sicstus Prolog*. Vienna, Austria, July 1998. URL `http://www.sics.se/isl/sicstus/sicstus_34.html`.

[55] Ilog CPLEX 6.0. Reference Manual. Ilog SA, Gentilly, France, 1998.

[56] Ilog OPL Studio 3.0. Reference Manual. Ilog SA, Gentilly, France, 2000.

[57] Ilog Solver 4.4. Reference Manual. Ilog SA, Gentilly, France, 1998.

[58] J.-L. J. Imbert. Linear constraint solving in clp-languages. In A. Podelski, editor, *Constraint Programming: Basics and Trends*, LNCS 910. Springer, 1995.

[59] S. Janson. *AKL - A Multiparadigm Programming Language*. PhD thesis, SICS Swedish Institute of Computer Science, SICS Box 1263, S-164 28 Kista, Sweden, 1994. SICS Dissertation Series 14.

[60] S. Janson and S. Haridi. Programming paradigms of the Andorra kernel language. In V. Saraswat and K. Ueda, editors, *Logic Programming, Proceedings of the 1991 International Symposium*, pages 167–186, San Diego, CA, USA, Oct. 1991. The MIT Press.

[61] S. Janson, J. Montelius, and S. Haridi. Ports for objects. In *Research Directions in Concurrent Object-Oriented Programming*. The MIT Press, Cambridge, MA, USA, 1993.

[62] F. Laburthe and Y. Caseau. SALSA: A Language for Search Algorithms. In *Fourth International Conference on the Principles and Practice of Constraint Programming (CP'98)*, Pisa, Italy, October 1998.

[63] W. Leler. *Constraint programming languages: their specification and generation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988. ISBN 0-201-06243-7.

[64] M. J. Maher. Logic semantics for a class of committed-choice programs. In J.-L. Lassez, editor, *4th International Conference on Logic Programming*, pages 858–876, Cambridge, Mass., 1987. MIT Press.

[65] K. Marriott and P. J. Stuckey. *Programming with Constraints: An Introduction*. MIT Press, Cambridge, Mass., 1998.

[66] L. Menezes, J. Vitorino, and M. Aurelio. A High Performance CHRv Execution Engine. In *Second Workshop on Constraint Handling Rules, at ICLP05*, Sitges, Spain, October 2005.

[67] S. Menju, K. Sakai, Y. Sato, and A. Aiba. A study on boolean constraint solvers. In F. Benhamou and A. Colmerauer, editors, *Constraint Logic Programming: Selected Research*, pages 253–268. MIT Press, Cambridge, Mass., 1993.

[68] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theor. Comput. Sci.*, 96(1):73–155, 1992. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/0304-3975(92)90182-F.

[69] P. Meseguer. Interleaved Depth-First Search. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, Nagoya, Japan, August 1997.

[70] L. Michel and P. Van Hentenryck. A Constraint-Based Architecture for Local Search. In *Conference on Object-Oriented Programming Systems, Languages, and Applications.*, pages 101–110, Seattle, WA, USA, November 4-8 2002. ACM.

[71] L. Michel and P. Van Hentenryck. A Modeling Layer for Constraint-Programming Libraries. *INFORMS Journal on Computing*, 2004. in press.

[72] L. Michel and P. Van Hentenryck. Non-deterministic control for hybrid search. In *CPAIOR'05: Proceedings of the 2nd International Conference on the Integration of Constraint Programming, Artificial Intelligence and Operations Research"*, pages 1–15, Prague, Czech Republic, 2005. Springer-Verlag.

[73] R. Mohr and G. Masini. Good old discrete relaxation. In *8th European Conference on Artificial Intelligence*, pages 651–656, Munich, Germany, 1988.

[74] J. Montelius. *Exploiting Fine-grain Parallelism in Concurrent Constraint Languages*. PhD thesis, SICS Swedish Institute of Computer Science, SICS Box 1263, S-164 28 Kista, Sweden, Apr. 1997. SICS Dissertation Series 25.

[75] J. Montelius and K. A. M. Ali. An And/Or-parallel implementation of AKL. *New Generation Computing*, 13–14, Aug. 1995.

[76] Mozart Consortium. The Mozart programming system, 1999. Available from `www.mozart-oz.org`.

[77] L. Perron. Search procedures and parallelism in constraint programming. In *CP '99: Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming*, pages 346–360, London, UK, 1999. Springer-Verlag. ISBN 3-540-66626-5.

[78] J.-F. Puget. A C++ Implementation of CLP. In *Proceedings of SPICIS'94*, Singapore, November 1994.

[79] J.-F. Puget and M. Leconte. Beyond the Glass Box: Constraints as Objects. In *Proceedings of the International Symposium on Logic Programming (ILPS-95)*, pages 513–527, Portland, OR, November 1995.

[80] V. Saraswat. *Concurrent Constraint Programming*. MIT Press, Cambridge, Mass., 1993.

[81] V. A. Saraswat. *Concurrent Constraint Programming*. ACM Doctoral Dissertation Awards: Logic Programming. The MIT Press, Cambridge, MA, USA, 1993.

[82] V. A. Saraswat and M. Rinard. Concurrent constraint programming. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 232–245, New York, NY, USA, 1990. ACM Press. ISBN 0-89791-343-4. doi: http://doi.acm.org/10.1145/96709.96733.

[83] V. A. Saraswat, M. Rinard, and P. Panangaden. The semantic foundations of concurrent constraint programming. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 333–352, New York, NY, USA, 1991. ACM Press. ISBN 0-89791-419-8. doi: http://doi.acm.org/10.1145/99583.99627.

[84] T. Schrijvers. Analyses, optimizations and extensions of constraint handling rules, Ph.D. Thesis. Technical report, Department of Computer Science, K.U.Leuven, Belgium, June 2005.

[85] T. Schrijvers and T. Frühwirth. Optimal union-find in constraint handling rules, programming pearl. *Theory and Practice of Logic Programming (TPLP)*, 6(1), 2006. URL `http://arxiv.org/abs/cs.PL/0501073`.

[86] T. Schrijvers and T. Frühwirth. CHR Website, `www.cs.kuleuven.ac.be/~dtai/projects/CHR/`, 2006.

[87] T. Schrijvers, B. Demoen, G. Duck, P. Stuckey, and T. Frühwirth. Automatic implication checking for chr constraints. In *6th International Workshop on Rule-Based Programming*, Apr. 2005. URL `http://www.cs.kuleuven.ac.be/~dtai/publications/files/41606.pdf`.

[88] T. Schrijvers, P. J. Stuckey, and G. J. Duck. Abstract interpretation for constraint handling rules. In *PPDP '05: Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 218–229, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-090-6. doi: http://doi.acm.org/10.1145/1069774.1069795.

[89] T. Schrijvers, B. Demoen, G. Duck, P. Stuckey, and T. Frühwirth. Automatic Implication Checking for CHR Constraints. *Electronic Notes in Theoretical Computer Science, Proceedings of 6th International Workshop on Rule-Based Programming, Nara, Japan, 2005*, 147(1):93–111, January 2006.

[90] C. Schulte. Programming deep concurrent constraint combinators. In E. Pontelli and V. S. Costa, editors, *Practical Aspects of Declarative Languages, Second International Workshop, PADL 2000*, volume 1753 of *Lecture Notes in Computer Science*, pages 215–229, Boston, MA, USA, Jan. 2000. Springer-Verlag.

[91] C. Schulte. Parallel search made simple. In N. Beldiceanu, W. Harvey, M. Henz, F. Laburthe, E. Monfroy, T. Müller, L. Perron, and C. Schulte, editors, *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000*, number TRA9/00, pages 41–57, 55 Science Drive 2, Singapore 117599, Sept. 2000.

[92] C. Schulte. *Programming Constraint Services*, volume 2302 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, Berlin, Germany, 2002.

[93] C. Schulte. Oz Explorer: A visual constraint programming tool. In L. Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 286–300, Leuven, Belgium, July 1997. The MIT Press.

[94] C. Schulte. Programming constraint inference engines. In G. Smolka, editor, *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming*, volume 1330 of *Lecture Notes in Computer Science*, pages 519–533, Schloß Hagenberg, Linz, Austria, Oct. 1997. Springer-Verlag.

[95] E. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):413–510, 1989.

[96] E. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):413–510, 1989.

[97] G. A. Sidebottom. A language for optimizing constraint propagation, Ph.D. Thesis. Technical report, Simon Fraser University, Canada, 1993.

[98] G. Smolka. The Oz programming model. In J. van Leeuwen, editor, *Computer Science Today*, LNCS 1000, Berlin, Heidelberg, New York, 1995. Springer.

[99] G. Smolka. A foundation for higher-order concurrent constraint programming. In J.-P. Jouannaud, editor, *1st International Conference on Constraints in Computational Logics*, volume 845 of *Lecture Notes in Computer Science*, pages 50–72, München, Germany, Sept. 1994. Springer-Verlag.

[100] G. Smolka. The Oz programming model. In J. van Leeuwen, editor, *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 324–343. Springer-Verlag, Berlin, 1995.

[101] J. Sneyers, T. Schrijvers, and B. Demoen. The Computational Power and Complexity of Constraint Handling Rules. In *Second Workshop on Constraint Handling Rules, at ICLP05*, Sitges, Spain, October 2005.

[102] K. Ueda. Guarded horn clauses. In *Concurrent Prolog*, pages 140–156, Cambridge, MA, USA, 1988. MIT Press. ISBN 0-262-19255-1.

[103] P. Van Hentenryck. Constraint and Integer Programming in OPL. *Informs Journal on Computing*, 14(4):345–372, 2002.

[104] P. Van Hentenryck. *The OPL Optimization Programming Language*. The MIT Press, Cambridge, Mass., 1999.

[105] P. Van Hentenryck and L. Michel. Nondeterministic Control For Hybrid Search. In *Proceedings of the Second International Conference on the Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems (CP-AI-OR'04)*, Prague, Czech Republic, 2005. Springer-Verlag.

[106] P. Van Hentenryck and L. Michel. *Constraint-Based Local Search*. The MIT Press, Cambridge, Mass., 2005.

[107] P. Van Hentenryck and L. Michel. *New Trends in Constraints*, chapter OPL Script: Composing and Controlling Models. Lecture Note in Artificial Intelligence (LNAI 1865). Springer Verlag, 2000.

[108] P. van Hentenryck, Y. Deville, and C.-M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 1992.

[109] P. van Hentenryck, V. A. Saraswat, and Y. Deville. Constraint processing in cc(FD). In A. Podelski, editor, *Constraint Programming: Basics and Trends*, LNCS 910. Springer, 1995.

[110] P. Van Hentenryck, L. Michel, and Y. Deville. *Numerica: a Modeling Language for Global Optimization*. The MIT Press, Cambridge, Mass., 1997.

[111] P. Van Hentenryck, L. Michel, and F. Benhamou. Newton: Constraint programming over nonlinear constraints. *Sci. Comput. Program.*, 30(1-2):83–118, 1998. ISSN 0167-6423. doi: http://dx.doi.org/10.1016/S0167-6423(97)00008-7.

[112] P. Van Hentenryck, L. Perron, and J.-F. Puget. Search and Strategies in OPL. *ACM Transactions on Computational Logic*, 1(2):1–36, October 2000.

[113] P. Van Hentenryck, L. Michel, F. Paulin, and J. Puget. *Modeling Languages in Mathematical Optimization*, chapter The OPL Studio Modeling System. Kluwer Academic Publishers, 2003.

[114] P. Van Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming*. The MIT Press, Cambridge, MA, USA, 2004.

[115] P. Van Roy, S. Haridi, P. Brand, G. Smolka, M. Mehl, and R. Scheidhauer. Mobile objects in Distributed Oz. *ACM Transactions on Programming Languages and Systems*, 19(5):804–851, Sept. 1997.

[116] P. Van Roy, P. Brand, D. Duchier, S. Haridi, M. Henz, and C. Schulte. Logic programming in the context of multiparadigm programming: the Oz experience. *Theory and Practice of Logic Programming*, 3(6):715–763, Nov. 2003.

[117] P. V. Weert, T. Schrijvers, and B. Demoen. The K.U.Leuven JCHR System. In *Second Workshop on Constraint Handling Rules, at ICLP05*, Sitges, Spain, October 2005.

[118] A. Wolf. Adaptive Constraint Handling with CHR in Java. In *7th International Conference on Principles and Practice of Constraint Programming (CP 2001)*, LNCS 2239. Springer, 2001.

Appendices