Artificial Intelligence 58 (1992) 113–159
Elsevier

ARTINT 951

Constraint satisfaction using constraint logic programming

Pascal Van Hentenryck

Brown University, Box 1910, Providence, RI 02912, USA

Helmut Simonis and Mehmet Dincbas

Cosytec, Parc Club Orsay-University, 4, rue Jean-Rostand, 91893 Orsay Cedex, France

Abstract

Van Hentenryck, P., H. Simonis and M. Dincbas, Constraint satisfaction using constraint logic programming, Artificial Intelligence 58 (1992) 113–159.

Constraint logic programming (CLP) is a new class of declarative programming languages whose primitive operations are based on constraints (e.g. constraint solving and constraint entailment). CLP languages naturally combine constraint propagation with nondeterministic choices. As a consequence, they are particularly appropriate for solving a variety of combinatorial search problems, using the global search paradigm, with short development time and efficiency comparable to procedural tools based on the same approach. In this paper, we describe how the CLP language cc(FD), a successor of CHIP using consistency techniques over finite domains, can be used to solve two practical applications: test-pattern generation and car sequencing. For both applications, we present the cc(FD) program, describe how constraint solving is performed, report experimental results, and compare the approach with existing tools.

1. Introduction

The purpose of our research is to support, within constraint programming languages, computational paradigms underlying combinatorial search problems. It is motivated by the hope of reducing significantly the develop-

Correspondence to: P. Van Hentenryck, Brown University, Department of Computer Science, 115 Waterman St., 4th floor, Providence, RI 02906, USA. E-mail: pvh@cs.brown.edu.

ment time of these applications while preserving most of the efficiency of procedural languages.

Combinatorial problems are ubiquitous in computer science. They appear in areas as diverse as operations research (e.g. scheduling), hardware design (e.g. circuit verification), biology (e.g. DNA sequencing), finance (e.g. option trading), and software design (e.g. simulation and testing of protocols), to name a few. Many of these problems are of high complexity (NP-complete or worse), which means that there is no efficient algorithm for solving them. Much research, however, has been spent on designing algorithms to tackle these problems and one of the interesting outcomes has been the development of constraint solving algorithms for various classes of problems.

Constraint programming has a long tradition in artificial intelligence. It can be traced back to the use of constraints in Sutherland's SKETCHPAD [67], the CONSTRAINT programming language of Sussman and Steele [66] and the work of Borning on ThingLab [2] among others. Mackworth also advocated, as early as 1977, the use of consistency techniques (a paradigm emerging from artificial intelligence to solve combinatorial search problems) in declarative languages as an alternative to chronological backtracking [42]. Constraint processing itself has also been present in many systems related to constraint solving such as REF-ARF [23], Alice [40], (assumption-based) truth maintenance systems (e.g. [15,21]), and various scheduling and planning systems (e.g. [24]).

The starting point of our research was, however, slightly different. We began by recognizing that logic programming is an appropriate language for stating combinatorial search problems: its relational form makes it easy to state constraints while its (don't-know) nondeterminism removes the need for programming a search procedure. Unfortunately, traditional logic programming languages can also be very inefficient when presented with a natural formulation of combinatorial search problems, largely because of their passive use of constraints to test potential values instead of pruning the search space in an active manner [27]. As a consequence, traditional logic programming languages (e.g. Prolog) often lead to "generate and test" or "standard backtracking" approaches that exhibit the pathological behavior known as thrashing [42].

Early (CLP) languages such as CHIP [20], CLP($\mathcal{R}$) [37], Prolog II [13], and Prolog III [12] attempted to preserve the advantages of logic programming while removing their limitations. The fundamental idea behind these languages, to use constraint solving instead of unification as the kernel operation of the language, was elegantly captured in the CLP scheme [36]. The CLP scheme defines a family of programming languages based on constraint solving and sharing the same semantic properties. It can be instantiated to produce a specific language by defining a constraint system

(i.e. defining a set of primitive constraints and providing a constraint solver for the constraints). Thus CHIP contains constraint systems over finite domains [72], Booleans [6], and rational numbers [30,74], Prolog III is endowed with constraint systems over Booleans, rational numbers, and lists, while CLP($\mathcal{R}$) solves constraints over real numbers. The CLP scheme was further generalized into the cc framework of concurrent constraint programming [54–56] to accommodate additional constraint operations (e.g. constraint entailment [43]) and new ways of combining them (e.g. implication or blocking ask [54] and cardinality [73]). More precisely, the cc framework accommodates all operations on constraints that can be defined as closure operators. The generalization significantly extends the scope of CLP languages by enabling issues such as concurrency, control, and extensibility to be addressed at the language level.

CLP languages[1] support, in a declarative way, the solving of combinatorial search problems using the global search paradigm. The global search paradigm amounts to recursively dividing a problem into subproblems until the subproblems are simple enough to be solved in a straightforward way, and includes, as special cases, implicit enumeration, branch and bound, and constraint satisfaction. It is best contrasted with the local search paradigm, which proceeds by modifying an initial configuration locally until a solution is obtained. These approaches are orthogonal and complementary. The global search paradigm has been used successfully to solve a large variety of combinatorial search problems with reasonable efficiency (e.g. scheduling [7], graph coloring [39], Hamiltonian circuits [9], and microcode labeling [19]) and provides, at the same time, the basis for exact methods as well as approximate solutions (giving rise to the so-called "anytime algorithms" [14]).

The purpose of this paper is to illustrate how CLP languages can be used to solve two practical combinatorial search problems: test-pattern generation and car sequencing. Test-pattern generation is a standard problem in hardware design and many algorithms have been proposed for the task. We show how to use constraint logic programming to design a simple algorithm whose behavior is similar in spirit to some of the best algorithms for the task and whose efficiency is competitive with specialized implementations of these algorithms. The second problem, car sequencing, was motivated by its presentation as a challenge for AI tools [48,49]. We propose a solution to this problem that can be described concisely in constraint logic programming and whose efficiency enables to solve large instances.

The CLP language used in the above problems is cc(FD), an instance of the cc framework over finite domains that is best seen as a successor to the

[1] In the following, we use the term *CLP languages* generically to denote both CLP and cc languages.

finite-domain part of CHIP. Both languages support the use of consistency techniques and local propagation in conjunction with don't-know nondeterminism approximated by backtracking. In addition, they support depth-first branch and bound for combinatorial optimization problems. The novel aspects of cc(FD) include the definition of new general-purpose combinators (such as cardinality, implication, constructive disjunction, and indexical constraints) and the availability of constraint entailment and constraint generalization as primitive operations on constraints. cc(FD) generalizes in an elegant way (and thus makes unnecessary) several features and constraints of CHIP that were difficult to justify theoretically. As a consequence, it provides additional operational expressiveness, flexibility, and efficiency and lets us tackle problems such as disjunctions of constraints and the definition of primitive constraints. Preliminary solutions of the two problems described here were first expressed in CHIP (see [60] and [18]). The presentation proposed in this paper subsumes them, both in the algorithmic methods, which are more advanced, and in the statement, which is simpler, more natural, and based on a solid theoretical foundation.

The rest of the paper is organized as follows. Section 2 presents a tutorial overview of cc(FD). Since the focus here is on applications, this overview is limited to those aspects of direct relevance to the two problems considered. Important combinators such as constructive disjunctions and indexical constraints are omitted here but can be found in [75]. Sections 3 and 4 present respectively the test generation and car sequencing problems. For each application, we describe in detail how the problem can be stated and how constraint solving is performed and we also report a number of experimental results and comparisons. Section 5 contains our conclusions.

2. Overview of cc(FD)

Here we give an informal overview of the relevant parts of cc(FD). A more formal presentation, following the style of operational semantics in [54], is given in the appendix.

Our overview proceeds in several steps. Section 2.1 sketches the syntax of the language and Section 2.2 introduces the CLP scheme. Sections 2.3, 2.4, and 2.5 discuss constraint entailment, the implication combinator, and the cardinality combinator, and Section 2.6 discusses the details of constraint solving in cc(FD). Note that the presentation separates the generic aspects of the language from the details of its constraint solver. This indicates that the combinators are general-purpose.

2.1. Syntax

Figure 1 shows an outline of the syntax of a cc(FD) program. A cc(FD)

```
Program  ::=  Clauses
Clauses  ::=  Head :- Body | Clauses Clauses
Head     ::=  Atom
Goal     ::=  Atom
Body     ::=  true | Goal | c | Body ,
              Body | c → Body | #(l,u,[c1,...,cn])
```

Fig. 1. An outline of the syntax.

```
p(X,Y,X) :-
  X ∈ {0,...,10}, Y ∈ {0,...,10}, Z ∈ {0,...,10},
  X ⩾ Z + 3,
  Y ⩽ Z,
  q(X,Y,Z).

q(X,Y,Z) :-
  r(X,Y).
q(X,Y,Z) :-
  Z ⩾ Y + 2.

r(X,Y) :-
  X ⩽ Y + 2.
```

Fig. 2. A simple program.

is a set of clauses in which each clause has a head and a body. A head is an atom, i.e. an expression of the form `p(t1,...,tn)` where `t1,...,tn` are terms. A term is a variable (e.g. `X`) or a function symbol of arity n applied to n terms (e.g. `f(X,g(Y))`). A body is either `true` (the empty body), a goal (procedure call), a constraint (constraint solving), an implication, or a cardinality combinator. In this paper, variables are denoted by uppercase letters, constraints by the letter c, conjunctions of constraints by the letter σ, terms by letters t and s, atoms by letters H and B, goals by the letter G, and integers by the letters l, u, and v, all possibly subscripted or superscripted. We also use C to denote a constraint system and D, possibly subscripted, to denote a finite domain. To illustrate the operational semantics of (part of) cc(FD), we use the simple program depicted in Fig. 2.

2.2. The CLP scheme

At least from a conceptual standpoint, the operational semantics of the CLP scheme is a simple generalization of the semantics of logic programming. It can be described as a goal-directed derivation procedure from the initial goal using the program clauses. A *computation state* is best described

by

(1) a *goal part*: the conjunction of goals to be solved;
(2) a *constraint store*: the set of constraints accumulated so far.

Initially the constraint store is empty and the goal part is the initial goal. In the following, we denote the computation state by pairs $\langle G \square \sigma \rangle$, where G is the goal part and σ is the constraint store. We use ε to denote an empty goal part or constraint store. An example computation state is

$$\langle \mathtt{q(X,Y,Z)} \square \mathtt{X,Y,Z} \in \mathtt{\{0,...,10\}} \ \& \ \mathtt{X} \geqslant \mathtt{Z} + 3 \ \& \ \mathtt{Y} \leqslant \mathtt{Z} \rangle.$$

A *computation step* (i.e. the transition from one computation state to another) can be of two types depending upon the selection of an atom or a constraint in the goal part. In the first case, a computation step amounts to

(1) selecting an atom in the goal part;
(2) finding a clause that can be used to resolve the atom; this clause must have the same predicate symbol as the atom, and the equality constraints between the goal and head arguments must be consistent with the constraint store;
(3) defining the new computation state as the old one where the selected atom has been replaced by the body of the clause and the equality constraints have been added to the constraint store.

In the second case, a computation step amounts to

(1) selecting a constraint in the goal part that can be satisfied with the constraint store;
(2) defining the new computation state as the old one where the selected constraint has been removed from the goal part and added to the constraint store.

For instance, given a computation state

$$\langle \mathtt{q(X,Y,Z)} \square \mathtt{X,Y,Z} \in \mathtt{\{0,...,10\}} \ \& \ \mathtt{X} \geqslant \mathtt{Z} + 3 \ \& \ \mathtt{Y} \leqslant \mathtt{Z} \rangle$$

a computation step can be performed using the second clause of q (see Fig. 2) to obtain a new computation state

$$\langle \mathtt{Z} \geqslant \mathtt{Y} + 2 \square \mathtt{X,Y,Z} \in \mathtt{\{0,...,10\}} \ \& \ \mathtt{X} \geqslant \mathtt{Z} + 3 \ \& \ \mathtt{Y} \leqslant \mathtt{Z} \rangle.$$

Another computation step leads to the configuration

$$\langle \varepsilon \square \mathtt{X,Y,Z} \in \mathtt{\{0,...,10\}} \ \& \ \mathtt{X} \geqslant \mathtt{Z} + 3 \ \& \ \mathtt{Y} \leqslant \mathtt{Z} \ \& \ \mathtt{Z} \geqslant \mathtt{Y} + 2 \rangle,$$

since the resulting constraint store is satisfiable. Note that, strictly speaking, equations should have appeared between the variables in the above example; they were omitted for clarity, since the variables have the same names in the program.

As should be clear, the basic operation of the language amounts to deciding the satisfiability of a conjunction of constraints. Note also that each computation state has a satisfiable constraint store. This property is exploited inside CLP languages to avoid solving the satisfiability problem from scratch at each step. Instead, CLP languages keep a reduced (e.g. solved) form of the constraints and transform the existing solution into a solution including the new constraints. Hence the constraint solver is made incremental. For instance, the last constraint store may be represented as

$$\langle \varepsilon \;\square\; X \in \{5,\ldots,10\} \;\&\; Y \in \{0,\ldots,5\} \;\&\; Z \in \{2,\ldots,7\} \;\&\; X \geqslant Z + 3 \;\&\; Y \leqslant Z \;\&\; Z \geqslant Y + 2 \rangle.$$

A computation state is *terminal* if

- the goal part is empty;
- no clause can be applied to the selected atom to produce a new computation state or the selected constraint cannot be satisfied with the constraint store.

A *computation* is simply a sequence of computation steps that either ends in a terminal computation state or diverges. A finite computation is *successful* if the final computation state has an empty goal, and *fails* otherwise.

To illustrate computations in a CLP language, consider our simple program again. The program has only one successful computation, namely

$$\langle p(X,Y,Z) \;\square\; \varepsilon \rangle$$

$$\downarrow \text{ (selecting the first constraint)}$$

$$\ldots$$

$$\downarrow \text{ (selecting the last constraint)}$$

$$\langle q(X,Y,Z) \;\square\; X,Y,Z \in \{0,\ldots,10\} \;\&\; X \geqslant Z + 3 \;\&\; Y \leqslant Z \rangle$$

$$\downarrow \text{ (using the second clause of q)}$$

$$\langle Z \geqslant Y + 2 \;\square\; X,Y,Z \in \{0,\ldots,10\} \;\&\; X \geqslant Z + 3 \;\&\; Y \leqslant Z \rangle$$

$$\downarrow \text{ (selecting the constraint)}$$

$$\langle \varepsilon \;\square\; X,Y,Z \in \{0,\ldots,10\} \;\&\; X \geqslant Z + 3 \;\&\; Y \leqslant Z \;\&\; Z \geqslant Y + 2 \rangle$$

The program has also one failed computation:

$\langle$ p(X,Y,Z) $\square$ ε $\rangle$

$\downarrow$ (selecting the first constraint)

...

$\downarrow$ (selecting the last constraint)

$\langle$ q(X,Y,Z) $\square$ X,Y,Z $\in$ {0,...,10} & X $\geqslant$ Z + 3 & Y $\leqslant$ Z $\rangle$

$\downarrow$ (using the first clause of q)

$\langle$ r(X,Y,Z) $\square$ X,Y,Z $\in$ {0,...,10} & X $\geqslant$ Z + 3 & Y $\leqslant$ Z $\rangle$

$\downarrow$ (using the clause of r)

$\langle$ X $\leqslant$ Y + 2 $\square$ X,Y,Z $\in$ {0,...,10} & X $\geqslant$ Z + 3 & Y $\leqslant$ Z $\rangle$.

The last computation state is terminal since the conjunction of constraints

X $\geqslant$ Z + 3 & Y $\leqslant$ Z & X $\leqslant$ Y + 2

is not satisfiable.

Note that the results of the computation are the constraint stores of the successful computations. Also, nothing has been said so far on the strategy used to explore the space of computations. Most CLP languages use a computation model similar to Prolog: atoms are selected from left to right in the clauses, clauses are tried in textual order, and the search space is explored in a depth-first manner with chronological backtracking in case of failures.[2] For instance, on the simple program, a CLP language typically uses the first clause for p, then the first clause for q, and finally encounters a failure when trying to solve r. Execution then backtracks to the second clause of q, giving the successful computation.

2.3. *Constraint entailment*

As mentioned previously, the cc framework considers other operations on constraints beyond constraint solving as well as additional ways of combining them. An important operation on constraints is *constraint entailment*, which amounts to finding out if a single constraint is implied by a conjunction of constraints, i.e.

$$\mathcal{C} \models (\forall)(\sigma \rightarrow c).$$

Constraint entailment was introduced in the context of concurrent logic programming (e.g. [58]) by Maher [43] to endow these languages with a logical semantics. It can be viewed as well as a generalization of languages allowing coroutining and delay mechanisms (e.g. [10,13,17,28,47]), and is

[2]We see below that the additional combinators of cc(FD) permit more sophisticated search procedures.

one of the cornerstones of the cc framework, where it is used to synchronize concurrently executing agents. It was also used in CHIP (see [20,31]) inside the `if_then_else` construct and was instrumental in simulating hybrid circuits. Its interest for CLP languages lies in the opportunity it gives to reason about the constraints and to use the information gained in pruning. As we will see, it can be used to express non-primitive constraints following general principles from artificial intelligence and operations research.

Both implication and cardinality, the two cc(FD) combinators used in our applications, make use of constraint entailment. The implication combinator was introduced in [54] in the context of concurrent logic programming, while the cardinality combinator was proposed explicitly for CLP languages in [73].

2.4. The implication combinator

Motivation

Local propagation is one of the key ideas behind constraint programming languages such as CONSTRAINTS [66] and ThingLab [2]. Local propagation (or value propagation) amounts to deducing values for some variables from those of other variables. For instance, an "and-gate" in a digital circuit may be defined by rules of the form

> "If one input is 0 then the output is 0",
> "If the output is 1 then the inputs are both 1".

To implement a program achieving this form of propagation, it is necessary to introduce a form of data-driven computation in which goals are suspended when not enough information is available and reactivated when new information allows them to be reconsidered. The purpose of the implication combinator for CLP languages is to achieve this form of behavior, to generalize it to any constraint system, and to combine it with nondeterministic choice.

Description

As mentioned previously, the implication combinator has the form $c \rightarrow A$ where c is a constraint and A is a body. Its declarative semantics is simply given by logical implication.

The main originality of the implication combinator lies in its operational semantics. The implication $c \rightarrow A$ ensures that A is executed only when (and as soon as) c is entailed by the constraint store. In other words, if c is entailed by the constraint store, $c \rightarrow A$ reduces to A. If $\neg c$ is entailed by the constraint store, $c \rightarrow A$ reduces to *true*. Otherwise, the computation blocks, waiting for more information.

Consider again the description of an and-gate using local propagation techniques:

```
and(X,Y,Z) :-
  X = 0 → Z = 0,
  Y = 0 → Z = 0,
  Z = 1 → (X = 1 , Y = 1),
  X = 1 → Y = Z,
  Y = 1 → X = Z,
  X = Y → X = Z.
```

The first rule says that, as soon as the constraint store entails `X = 0`, the constraint `Z = 0` must be added to the constraint store. Note that the last three rules actually do more than local value propagation; they also propagate symbolic equalities and one of them is conditional to a symbolic equality. Now the goal $\langle$ `and(X,Y,Z)` $\square$ `X = 0` $\rangle$ produces a constraint store `X = 0 & Z = 0`, since the goal $\langle$ `X = 0 → Z = 0` $\square$ `X = 0` $\rangle$ reduces to $\langle$ `Z = 0` $\square$ `X = 0` $\rangle$ and hence to the constraint store `X = 0 & Z = 0`. However the goal $\langle$ `and(X,Y,Z)` $\square$ $\varepsilon\rangle$ does not modify the constraint store, since none of the constraints in the implication constructs are entailed by the constraint store.

As mentioned previously, a goal that is blocked can be resumed when new information become available in the constraint store. Assume for instance the computation state

$$\langle \texttt{X = 0} \rightarrow \texttt{Z = 0 , T = 0} \rightarrow \texttt{X = 0} \;\square\; \texttt{T = 0}\rangle.$$

The first goal `X = 0 → Z = 0` blocks since `X = 0` is not entailed by the constraint store. But the second goal can be executed, leading eventually to the computation state

$$\langle \texttt{X = 0} \rightarrow \texttt{Z = 0} \;\square\; \texttt{X = 0 \& T = 0}\rangle.$$

Now `X = 0` is entailed by the constraint store and hence the first implication can be executed. The final constraint store will be `X = 0 & T = 0 & Z = 0`.

Now consider building a full-adder using logical gates:

```
fa(X,Y,Cin,S,C) :-
  and(X,Y,C1),
  xor(X,Y,S1),
  and(Cin,S1,C2),
  xor(Cin,S1,S),
  or(C1,C2,C).
```

In the above circuit, `X` and `Y` are two input bits, `Cin` is the carry-in, `S` is the result bit, and `C` is the carry-out. If we use the implication combinator to define all logical gates, the query `fa(X,Y,1,S,0)` produces the constraint store

```
X = 0 & Y = 0 & S = 1.
```

The reason is the following. Since the result of the or-gate is 0, its two inputs C1 and C2 must be 0. Since the second and-gate has output C2 equal to 0 and input Cin equal to 1, it follows that S1 must be 0, which implies that X and Y must be equal because of the first xor-gate. Since X and Y appear both as inputs in the same and-gate, they must be equal to its output C1, which is 0.

The implication combinator thus introduces a notion of coroutining between goals in the language, and the execution of goals can be interleaved in complex ways. Note that the goals synchronize by "asking" if some constraints are entailed by the constraint store and that a suspended goal can be resumed by a modification of the constraint store by other goals. Moreover, the implication combinator is not restricted to simple constraints, as illustrated above, but allows arbitrary constraints of the language.

2.5. The cardinality combinator

Motivation

The cardinality combinator is a declarative and relational operator, intended for the handling of general forms of disjunctions which often occur in practical applications. It can be used to enforce arc-consistency on any arbitrary finite-domain constraints (within the complexity bound of the optimal algorithm of [44]) but, as should be clear from the presentation, it is not limited to finite-domain constraints. The cardinality has been used in numerous applications including scheduling, assignment, Hamiltonian circuit, and warehouse location problems. It will be important in the car sequencing application.

Before entering into the description of the combinator, let us give an example to motivate the reader. Consider, for instance, a scheduling problem and assume that we face a disjunctive constraint between two tasks, i.e. the execution of the two tasks cannot overlap. Assume that S1 and S2 represent the starting dates of the tasks and D1 and D2 their durations, the constraint can be expressed as

```
disjunctive(S1,D1,S2,D2) :-
  S1 + D1 ⩽ S2.
disjunctive(S1,D1,S2,D2) :-
  S2 + D2 ⩽ S1.
```

Unfortunately the above constraint is nondeterministic and introduces choice points during the execution. The first alternative, i.e. the second task is scheduled after the first task, will be selected and its constraint will be added to the constraint store. Subsequent execution may lead to a failure and require this choice to be reconsidered. The second alternative, i.e. the first task is scheduled after the second task, will then be considered. In

general, it is better to postpone choices as long as possible. The above constraint can be used in two ways to achieve pruning: (1) if the maximal start date of S2 is smaller than the minimal start date of S1 added to D1, then the second task cannot be scheduled after the first task and (2) if the maximal start date of S1 is smaller than the minimal start date of S2 added to D2, then the first task cannot be scheduled after the second task. The cardinality combinator enables us to express this pruning in a natural way.

Description

As mentioned previously, the cardinality combinator has the form

$$\#(l, u, [c_1, \ldots, c_n])$$

where l and u are integers and $c_1, \ldots, c_n$ are constraints.

The declarative semantics is given as follows. $\#(l, u, [c_1, \ldots, c_n])$ is true iff the number of constraints c_i $(1 \leqslant i \leqslant n)$ satisfiable is not less than l and not more than u. It is false otherwise.

Note that this combinator is quite expressive. A conjunction $c_1 \wedge \cdots \wedge c_n$ can be expressed as $\#(n, *, [c_1, \cdots, c_n])$ where $*$ is a don't-care value, a disjunction $c_1 \vee \cdots \vee c_n$ as $\#(1, *, [c_1, \ldots, c_n])$, and a negation $\neg c$ as $\#(*, 0, [c])$. Other connectives such as equivalence $\Leftrightarrow$ can now be obtained easily. In the applications, we feel free to use the logical operators instead of the cardinality combinator when convenient.

Using the cardinality combinator, the disjunctive constraint can be implemented as follows:

```
disjunction(S1,D1,S2,D2) :-
  #(1,*,[S1 + D1 ⩽ S2, S2 + D2 ⩽ S1]).
```

Once again, the main interest of the cardinality combinator lies in its operational semantics. The combinator implements a principle well known in operations research and artificial intelligence: "infer simple constraints from difficult ones". The intuitive idea is to make sure that the cardinality combinator can be satisfied in some way. Moreover, if there is only one way to satisfy it, then the constraints necessary to satisfy it are introduced in the constraint store. Constraint entailment is used to check if there is a way to satisfy the constraint. In the disjunctive example, the system makes sure that either the first task can be scheduled before the second one or the second task can be scheduled before the first one (or both). If the constraint store makes it impossible to schedule the first task before the second, then a constraint forcing the second task to be scheduled first is added to the constraint store.

Consider a simple example:

```
⟨#(1,2,[X=4, Y=10]) & X>6 □ ε⟩
        ↓
⟨#(1,2,[X=4, Y=10]) □ X>6⟩
        ↓
⟨#(1,2,[Y=10]) □ X>6⟩
        ↓
X>6 & Y=10
```

This example contains a cardinality combinator requiring that `X = 4` or `Y = 10` be true. Initially neither these two constraints nor their negations are entailed by the constraint store, so the execution of the cardinality combinator blocks. The second goal `X > 6` is selected, which implies that $X \neq 4$ is entailed by the constraint store. There is now only one way to satisfy the cardinality combinator, i.e. adding the constraint `Y = 10` to the constraint store.

The cardinality combinator can be used to enforce arc-consistency on any binary constraint in time $O(ed^2)$, where e is the number of constraints and d is the size of the largest domain. Given a constraint $c(X, Y)$ with $X \in D_x$ and $Y \in D_y$, it is sufficient to generate for each value $v \in D_x$ a constraint of the form

$$X = v \quad \Leftrightarrow \quad Y \in D$$

where $D = \{w \in D_y \mid c(v, w)\}$ and vice versa for Y. The equivalence can be rewritten easily into two cardinality formulas. The optimal bounds of Mohr and Henderson [44] can be obtained by using counters to implement cardinality and entailment.

2.6. Constraint system

Here we give an informal presentation of the constraint part of cc(FD).

Syntax

Definition 2.1. An arithmetic term is defined inductively as follows:

(1) A variable is an arithmetic term.
(2) A natural number is an arithmetic term.
(3) $t_1 + t_2$, $t_1 * t_2$, and $t_1 - t_2$ are arithmetic terms if t_1 and t_2 are arithmetic terms.

The primitive constraints of the language are as follows:

Definition 2.2. A primitive constraint in cc(FD) can be of two forms:

(1) $x\ \delta_1\ \{v_1, \ldots, v_n\}$;
(2) $t_1\ \delta_2\ t_2$,

where x is a variable, $v_1, \ldots, v_n$ are natural numbers, $\delta_1 \in \{\in, \notin\}$, t_1 and t_2 are arithmetic terms, and $\delta_2 \in \{>, \geqslant, =, \neq, \leqslant, <\}$. Constraints of the first type are called domain and non-membership constraints respectively, while constraints of the second type are called arithmetic constraints.

Note that in cc(FD) each variable appearing in an arithmetic constraint must also occur in a domain constraint.

Constraint solving

There are various ways of implementing a constraint solver for the above constraints. Since the problem is decidable (because all variables must appear in a domain constraint), a decision procedure is possible for consistency and entailment. However, a complete constraint solver would necessarily require exponential time (unless P = NP). The approach taken in cc(FD) (and in CHIP as well) is to use consistency techniques instead and amounts to replacing constraint solving by arc-consistency and constraint entailment by arc-entailment.

Definition 2.3. A constraint $c(x_1, \ldots, x_n)$ is *arc-consistent* with respect to $D_1, \ldots, D_n$ if, for each variable x_i and value $v_i \in D_i$, there exist values $v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n$ in $D_1, \ldots, D_{i-1}, D_{i+1}, \ldots, D_n$ such that $c(v_1, \ldots, v_n)$ holds.

A set of constraints is arc-consistent with respect to a set of domains for its variables iff all constraints are arc-consistent with respect to the domains.

Definition 2.4. A constraint $c(x_1, \ldots, x_n)$ is *arc-entailed* by $D_1, \ldots, D_n$ iff, for all values $v_1, \ldots, v_n$ in $D_1, \ldots, D_n$, $c(x_1, \ldots, x_n)$ holds.

The operational semantics of the parts of cc(FD) presented in this paper can be understood informally as an instance of the generic scheme presented earlier in which consistency is replaced by the weaker notion of arc-consistency and entailment by the weaker notion of arc-entailment. Enforcing arc-consistency does not in general produce a decision procedure (see [16] however for subclasses having that property). In conjunction with nondeterminism, it produces the kind of languages advocated in [42]. Arc-consistency algorithms have been intensively studied [42,44,45,77] but with the primitive constraints considered in cc(FD), more efficient algorithms can be exhibited. For instance, with binary constraints, arc-consistency can be enforced in $O(ed)$ where e is the number of constraints and d is the size of the largest domain [16].

A formal semantics of cc(FD) in terms of the cc framework requires decision algorithms for constraint solving and entailment. The key idea is to

divide the primitive constraints into two classes: (1) basic constraints (those allowing an efficient decision procedure) and (2) non-basic constraints defined in terms of the combinators.[3] The main benefit of investigating the formal semantics has been the identification of a number of new combinators (e.g. constructive disjunction and indexical constraints) that support, at the language level, pruning principles previously hidden in the implementation.

3. Test-pattern generation

The first application we consider is in the field of digital circuit design: automatic test-pattern generation (ATPG). Problems from circuit design are useful in evaluating general problem solving techniques, since many special-purpose methods have been developed in this area and different approaches can be compared, using widely available benchmarks. CLP in general, and CHIP in particular, have been applied to a number of problems from digital circuit design, including formal verification [65], diagnosis [63], synthesis [64] as well as simulation of hybrid circuits [31]. The use of CLP for test generation has been discussed before [59,60,62]. The method described here is based on [61]. We show that cc(FD) allows a simple and declarative formulation of test generation as a constraint satisfaction problem. Moreover, by using the implication operator to define demons, it is possible to design an efficient test generation algorithm that requires only a fraction of the development effort necessary with conventional approaches.

3.1. Problem statement

VLSI chips are produced by complex processes in which errors can arise, hence only a certain percentage of chips will be error free. This *yield* varies with different circuit types and processes, but can be as low as a few percent for a new fabrication process. The manufacturer, on the other hand, wants to sell chips with a low *defect level*, i.e. a low percentage of faulty chips passing quality control. *Test generation* is the process of defining the tests to apply to a circuit in order to detects faults. Williams [78] has presented a model expressing the defect level as a function of yield and *fault coverage*, the percentage of all faults detected by testing. This model makes clear the necessity of finding a very high percentage of all faults in order to obtain a low defect level for a process with a low yield.

[3]These constraints need not be considered primitive constraints in the language, since they can be defined at the language level.

3.1.1. Fault models

Since many different physical failures can occur in a circuit, the only way to test for all possible faults is to test all circuit behaviors over time, which is clearly impractical. The principal idea of *structural testing* is to use knowledge about the structure of a circuit and the underlying technology to limit the number of cases we have to consider. There have been many attempts to describe what types of faults can occur in different technologies [1]. One of the earliest and still widely used fault models is the "stuck-at" model. This assumes that all faults lead to the situation where some signal in a circuit is permanently set to "1" or "0". The signal is then said to be "stuck-at 1" (`sa1`) or "stuck-at 0" (`sa0`). This fault model covers many, though not all, device faults inside a VLSI circuit. It has been shown that a test set that detects all single stuck-at faults also covers many other faults (with the exception of time-sensitive faults). Most test generation systems restrict themselves to the detection of single stuck-at faults at the logical gate level. We will use this model and, in the rest of the section, *fault coverage* should be understood as the percentage of all detected single stuck-at errors.

Note also that testing for stuck-at faults in a circuit does not require generating tests for each fault, as some faults are *covered* by other faults [51]. For instance, testing the output of an and-gate for `sa0` automatically tests the gate inputs for `sa0`. We can easily generate this more interesting *collapsed fault set* in a preprocessing step.

3.1.2. Test generation and fault simulation

The ATPG problem is conceptually split into two subproblems: *test generation* and *fault simulation*. *Test generation* entails finding a test that detects a certain fault for some component inside the circuit; *fault simulation* detects which faults are covered by a particular pattern. Often the two parts are intertwined and the whole process terminates when either a preset fault coverage is obtained or a time limit is exceeded. The presentation here is restricted to the test generation phase, which typically consists of three steps [3]:

- *Setup*: To test a fault at the output of a certain gate, it is necessary to ensure different behavior for the good and faulty circuits for this signal. This can be achieved by *controlling the gate*, i.e. by applying certain signals to the inputs of the gate. For instance, testing an and-gate for a stuck-at-zero fault requires us to set both inputs of the gate to 1.
- *Propagation*: It is clearly not enough to create an internal difference between the behavior of the good and the faulty circuit. This difference must be *observable* at some output of the circuit. The propagation step creates a *sensitized path* from the gate under test to some circuit output. In general, one or several symbolic values are introduced and

the propagation step amounts to propagating these symbolic values towards the primary outputs. The symbolic values represent the value or the negation of the value at the gate under test and indicate where the result of the test can be observed.

- *Justification*: The last step assigns values to all signals in the circuit in order to satisfy the conditions enforced by the setup and propagation steps. Generating a test basically amounts to finding an assignment of values for each of the primary inputs, that satisfies the constraints imposed by the setup and propagation steps on the signals throughout the circuit.

How these steps are implemented makes the difference between the various test generation algorithms.

3.2. Problem solution

In this section, we present the test generation program in cc(FD). We proceed in several steps. Section 3.2.1 discusses how circuits can be represented in logic programming. Section 3.2.2 shows how ATPG can be seen as a constraint satisfaction problem. Section 3.2.3 shows how to implement the basic elements as demons using the implication operator. Section 3.2.4 presents the basic test generation program, and Section 3.2.5 shows how heuristics can improve the algorithm efficiency.

3.2.1. Circuit description

Logic programming can be considered as a simple but powerful hardware description language. It supports in a natural way top-down development and mixing of various hierarchical levels of circuit description. In logic programming, a circuit can be specified by means of *clauses* that describe components and modules and the interactions between them. A general description of a full-adder can be given as follows:

```
fa(M,N,X,Y,Z,S,C) :-
  and(M,[1|N],X,Y,C1),
  xor(M,[2|N],X,Y,S1),
  fanout(M,[3|N],S1,S11,S12),
  and(M,[4|N],Z,S11,C2),
  xor(M,[5|N],Z,S12,S),
  or(M,[6|N],C1,C2,C).
```

For simulation, the definition of the basic elements `and`, `xor`, and `or` can be given by a set of ground clauses (the truth table definition). For instance, an and-gate can be expressed as

```
and(simul,N,0,0,0).
```

```
and(simul,N,0,1,0).
and(simul,N,1,0,0).
and(simul,N,1,1,1).
```

Here the first argument contains the operation mode (for instance, "test" for test generation, "simul" for circuit simulation or "time" for delay time computation) to distinguish between several user-defined operation modes. The second argument assigns a unique identifier to each part (module or basic component) of the circuit. Thus a hierarchical naming convention can be easily implemented. The other arguments are the inputs and outputs of the components. Note that no distinction is necessary between inputs and outputs. Multiple internal connections between components are represented by *fanout points*, since they are of special interest in test generation. In previous examples (see Section 2.5), connections were represented by shared logical variables.

The full-adder can now be used in other circuit descriptions and parameterized libraries of modules can be generated using hierarchical descriptions. This kind of hierarchical description of circuits follows the style of logic programming in top-down development: one can replace the description of a lower-level component without affecting the higher-level circuit definition. The same circuit description can be used in various applications including simulation, formal verification and fault diagnosis (see [62]). Similar ways of describing hardware in logic programming are reported in [11,22,32,33,68].

3.2.2. ATPG as a constraint satisfaction problem

Our strategy is based on treating the test generation problem as a consistent labeling problem. We use six symbolic values, `0`, `1`, `d`, `dnot`, `e`, and `enot`. The values *d* and *e* represent the value at the gate under test while *dnot* and *enot* represent their negations. The basic difference between *d*, *dnot*, and *e*, *enot* is in the way these values are propagated. *d* is assigned to the gate under test and the goal of test generation is to propagate *d* or *dnot* to a primary output so that the gate can be observed. Once a test has been found, it is sufficient to run the circuit with the test and to observe the value of the gate at a suitable primary output. The values *e* and *enot* are introduced because of fanout points: without the values *e* and *enot*, a fanout point would need to propagate a *d* or *dnot* value to all outputs. Since the values *d* and *dnot* impose severe constraints on the gates in order to propagate them towards the primary outputs, the algorithm may be unable to find a test in some cases. With the values *e* and *enot*, a fanout point propagates a *d* (or a *dnot*) on one output and an *e* (or an *enot*) on the other outputs. Since it devotes no effort to propagating *e* and *enot*, the algorithm avoids the

and	0	1	d	$\bar{d}$	e	$\bar{e}$
0	0	0	--	--	0	0
1	0	1	d	$\bar{d}$	e	$\bar{e}$
d	--	d	--	--	d	--
$\bar{d}$	--	$\bar{d}$	--	--	--	$\bar{d}$
e	0	e	d	--	e	0
$\bar{e}$	0	$\bar{e}$	--	$\bar{d}$	0	$\bar{e}$

Fig. 3. Definition of an and-gate in six-value logic.

xor	0	1	d	$\bar{d}$	e	$\bar{e}$
0	0	1	d	$\bar{d}$	e	$\bar{e}$
1	1	0	$\bar{d}$	d	$\bar{e}$	e
d	d	$\bar{d}$	--	--	--	--
$\bar{d}$	$\bar{d}$	d	--	--	--	--
e	e	$\bar{e}$	--	--	0	1
$\bar{e}$	$\bar{e}$	e	--	--	1	0

Fig. 4. Definition of an xor-gate in six-value logic.

not	0	1	d	$\bar{d}$	e	$\bar{e}$
	1	0	$\bar{d}$	d	$\bar{e}$	e

Fig. 5. Definition of a not-gate in six-value logic.

above-mentioned drawback. The resulting algorithm is complete (it finds a test if one exists), which is not the case for the algorithm using a five-value logic.

Figures 3–5 give the definitions of some gates (*dnot* and *enot* are represented by $\bar{d}$ and $\bar{e}$). These definitions are intended to propagate the values d and *dnot* towards the primary outputs and hence some input combinations are prohibited. Consider for example the and-gate. If an input is a value d, then the other input must be either 1 or e in order to propagate d to the output. The handling of the value *dnot* is similar. Note also that the values e and *enot* are not necessarily propagated to the output of the gate; this illustrates the main difference between the values $e, enot$ and the values $d, dnot$. The xor-gate is also interesting to analyze. As soon as an input is d or *dnot*, the other input must be 0 or 1 respectively. Note that a value d can thus be propagated as a *dnot* on the output. The not-gate is straightforward.

The possible values for a fanout point are given by the predicate definition in Fig. 6. Note especially how the value d is propagated: only one of the outputs is assigned to d, the other being given the value e. There are of course two possible ways of propagating d depending upon the output chosen.

Test generation is then performed by the following method. Variables

```
% fanout(Mode,Label,Stem,Branch1,Branch2)
fanout(M,N,0,0,0).
fanout(M,N,1,1,1).
fanout(M,N,d,d,e).
fanout(M,N,d,e,d).
fanout(M,N,dnot,dnot,enot).
fanout(M,N,dnot,enot,dnot).
fanout(M,N,e,e,e).
fanout(M,N,enot,enot,enot).
```

Fig. 6. Definition of a fanout in the six-value logic.

throughout the circuit are required to take one of the six signal values. In addition, the primary inputs can only take values 0 or 1. One primary output will have a *d* or *dnot* value and some others can have *e* or *enot* values. The circuit gates impose local constraints between their inputs and outputs (defined by the truth tables above). The gate under test will have a *d* as output and suitable inputs to control the gate. The key advantage of this description is that all constraints can be expressed just as local constraints. The existence of a *d*-path from the *gate under test* to a primary output is guaranteed by the constraints. This is the main difference from the classical ATPG algorithms [26,29,53], which use a five-value logic and rely on a global control strategy to create the *d*-path and choose between alternatives. Note also that the solution is not described algorithmically by changes to be applied to an empty assignment, but rather as a constraint satisfaction problem.

3.2.3. Gates as demons

A simple definition of the gates as truth tables would lead to an extremely inefficient program. For a better approach, we exploit two features of cc(FD): domain constraints and the implication operator. Each line in the circuit is associated with a variable constrained to take one of the six possible values. In addition, the primary inputs are constrained to be 0 or 1. The implication operator is then used to define a demon for each type of gate. The demons make sure that the gates propagate values as soon as possible and reduce the search space whenever possible by removing values from the variables. The demon definition is a generalization of that presented in the description of the implication operator. For instance, the demon for an and-gate is depicted in Fig. 7.

Note that the implications use both equations and non-membership and domain constraints to reduce the search space by removing variable values. Also, each implication solves the constraint, i.e. if an implication has been

```
and_demon(X,Y,Z):-
  X = 0 → (Z = 0, Y ∉ {d,dnot}),
  Y = 0 → (Z = 0, X ∉ {d,dnot}),
  Z = 1 → (X = 1, Y = 1),
  X = 1 → Y = Z,
  Y = 1 → X = Z,
  X = Y → (X = Z, X ∉ {d,dnot}),
  X = d → (Y ∈ {1,e}, Z = d),
  Y = d → (Y ∈ {1,e}, Z = d),
  X = dnot → (Y ∈ {1,enot}, Z = dnot),
  Y = dnot → (Y ∈ {1,enot}, Z = dnot),
  X = e → Y = enot → Z = 0,
  X = enot → Y = e → Z = 0.
```

Fig. 7. Implementation of an and-gate in the six-value logic.

```
and(test(Gate1,Fault),Gate,X,Y,d) :-  % this is the g.u.t.
  Gate1 = Gate,                       % to test the fault
  inverse(Fault,Setup),               % setup opposite value
  and(X,Y,Setup).                     % use the 0-1 demon

and(test(Gate1,Fault),Gate,X,Y,Z):-
  Gate1 ≠ Gate,                       % it is not the g.u.t.
  Z ∈ {0,1,d,dnot,e,enot},            % domain constraint
  and_demon(X,Y,Z).                   % use the six-valued demon
```

Fig. 8. The and-gate definition for ATPG.

applied, then *all* remaining values for its variables are valid. Finally, note that we do not enforce an assignment of the gate inputs in the case where the output takes the value 0. The constraint blocks until an assignment is made to an input either by propagation or by a labeling routine.

3.2.4. The basic ATGP program

We can now present the basic program.

Each type of gate is associated with a new procedure. Figure 8 illustrates the approach for an and-gate. Besides the inputs and output, the procedure receives two arguments: a term `test(Gate,Fault)`, which is the same for all gates, and a unique identifier for the gate. The term `test(Gate,Fault)` indicates which gate `Gate` is under test for a given fault `Fault`; for example, `test([2],1)` is used for testing a `sa1` fault at gate 2.

The procedure for each type of gate is defined by two clauses. The first clause handles the case of the gate under test (g.u.t.), recognized by the

```
% test(+,+,+,-):  generate test for output of Gate at Fault sa0 or sa1
%                the third arg is a list of inputvars of the circuit

test(Gate,Fault,Inputlist,Output):-
  domain_constraints(Inputlist,0,1),
  circuit(test(Gate,Fault),[],Inputlist,Output),
  labeling(Inputlist).

% labeling(+):  assign 0 or 1 to all inputs of the circuit

labeling([]).
labeling([X|T]):-
  member(X,[0,1]),
  labeling(T).
```

Fig. 9. ATPG program.

equality `Gate1 = Gate`, where `Gate1` is the unique identifier of the gate under test and `Gate` is the gate currently considered. The clause simply assigns the value d to the output. In addition, the clause controls the gate by stating a constraint on the inputs to produce the desired output. The desired output is obtained from the type of fault by the procedure `inverse` and the constraint is enforced using the 0–1 definition of the and-gate as described in Section 2.4. For example, an `sa0` fault for the and-gate would produce 1 as the desired output (i.e. `Setup` is 1) and the 0–1 and-gate is called with `X`, `Y`, and `Setup` as arguments. In this case, the 0–1 definition assigns `X` and `Y` to 1. The second clause handles the general case, i.e. when the gate under consideration is not the gate under test. The clause simply enforces a domain constraint for the output and calls the six-value definition.

The complete program for test generation is shown in Fig. 9. It uses a circuit description and the predicate definitions above. The first argument of `test` is the label of the gate to test, the second argument is the fault type to test, and the third argument must be instantiated to the list of variables for the primary inputs of the circuits (this list is assigned 0–1 values by the labeling routine). The predicate `domain_constraints` generates suitable domain constraints for the primary inputs to guarantee that they are given a 0–1 value. The second goal enforces the constraints associated with each gate relating its inputs to outputs. The last goal simply assigns values to the primary inputs. As usual in constraint programming, the generation phase is interleaved with the constraint propagation part at run time, although they are separated in the problem statement.

The algorithm described so far is complete, i.e. it finds a test pattern if one exists and fails otherwise. We now explore several ways of improving its efficiency.

3.2.5. Heuristics

The basic procedure described so far requires making many possible choices. To obtain an efficient system, it is necessary to develop heuristics that avoid making the *wrong* choices. In this section we describe some of the heuristics used in our test generation program. We show that this information can easily be added into the program.

Controllability and observability

When propagating a d-value from the fault to a primary output, no choices are needed as long as there is a unique path. When a d-value reaches a fanout stem however, the d-path can continue along any of the stems and we have to decide which one to follow. Several measures have been proposed to estimate the difficulty of finding a path from some point inside the circuit to an output [4]. This value, called an *observability* measure, can be precomputed in a preprocessing step. For each fanout point, we obtain an ordering for the fanout stems, and try to propagate the d-value along the path with highest observability first.

A similar measure estimates how difficult it is to set a point inside the circuit to a particular value, 0 or 1. This *controllability* is used to decide which values to assign to controlling inputs of xor-gates in the d-path. If it is easy to set a point to 0, we use this value; if not, we set it to 1.

Both controllability and observability are heuristic values. Since they are obtained by simple computations, for example ignoring reconvergent fanout, they give only hints on which values to test first, and do not eliminate the need for backtracking completely.

Labeling

The choice of an appropriate labeling routine is crucial for many constraint satisfaction problems but turns out not to be as important for test generation. We use a routine that assigns the variables in the order given, but chooses randomly between 0 and 1 for the first assignment. For most of the example circuits tested below, the labeling is done without any deep backtracking.

Limiting backtracking

Some faults in circuits can be untestable; they are *redundant*. The program may not be able to detect this in reasonable time. Tests for other faults can be very difficult to obtain. To avoid spending too long trying to find a test, we have to limit the search performed on any one fault. This can be done in two ways: one is to limit the number of backtrack steps performed in the search, the other is to limit the execution time spent on each case. Both methods are rather simple to add to the program.

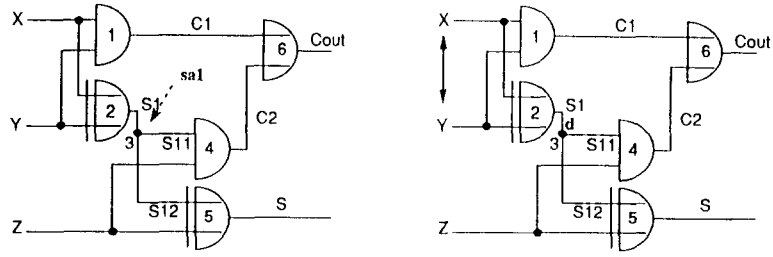

Fig. 10. Test generation example.

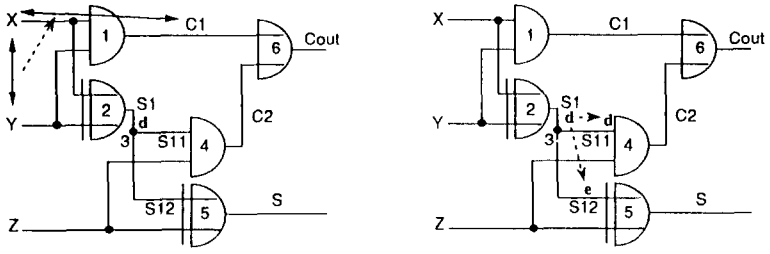

Fig. 11. Test generation example (continued).

3.3. Example

We use the full-adder circuit described above to illustrate the behavior of the program. We explain the steps required to generate a test for an `sa1` fault at the output of xor-gate 2 (see Fig. 10).

The query to execute is

```
?- test([2],1,[X,Y,Z],[S,Cout]).
```

The program enforces all constraints imposed by the circuit. To control gate 2, `S1` is assigned the value `d`. In addition, since the test is an `sa1` fault, the variable `Setup` is assigned to `0`. The xor-demon for the gate is then executed with the output equal to `0`. This assignment entails, by definition of the xor-gate, the equality of both inputs of gate 2 (see Fig. 10 where the equality is shown as a double arrow), which is the weakest constraint necessary to make sure that the output is `0`. All other gates use the six-value definitions; their purpose is to propagate the value `d` (or `dnot`) towards the primary outputs. Let us review how this is done.

The equality between `X` and `Y` enables one of the implications of the and-gate to be reduced (shown in the picture as a dotted arrow), leading to the equality of `X` with `C1` and the removal of `d` and `dnot` from `C1` (see Fig. 11). Then the rule for fanout point 3 is executed, creating the constraints `S11` = `d` and `S12` = `e` (see Fig. 11). This triggers another implication for and-gate 4, binding `Z` to `1` and `C2` to `d`, which in turn triggers an implication for gate 5, binding `S` to `enot` (see Fig. 12). The rule for gate 6 now binds `Cout` to `d` and `C1` to `0` and, by unification, `X` to `0` and `Y` to `0` (see Fig. 12).

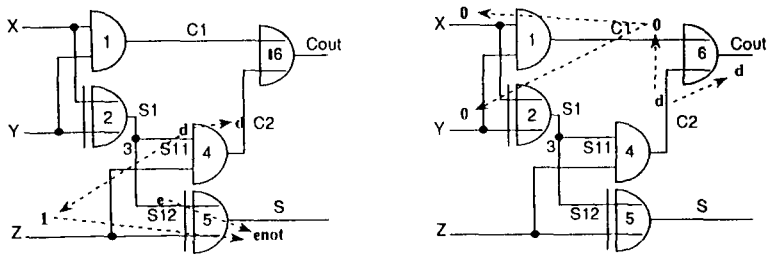

Fig. 12. Test generation example (continued).

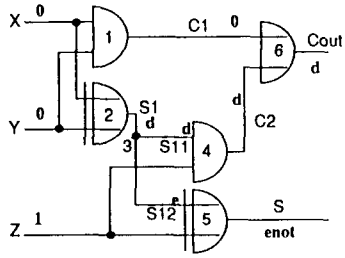

Fig. 13. Test generation example (continued).

The final solution is then

```
X = Y = C1 = 0,     S1 = d,
S11 = d,            S12 = e,
C2 = d,             Cout = d,
Z = 1,              S = enot
```

(as shown in Fig. 13). The test pattern generated for the `sa1` fault at the output of gate 2 is [0,0,1].

This example is unusual in that all constraints are ultimately solved, i.e. all variables are instantiated to values. No generation of values for the primary inputs is thus necessary. For more complex examples however, this will not be the case: some constraints will block and wait until variables are instantiated by the labeling procedure in the test predicate.

3.4. Computation results

An evaluation of a test generation method must include experiments with large, realistic circuits. We use the ISCAS benchmark set [35] to test our method. The results show that a constraint-based ATPG system, while currently not as fast as specialized programs, finds test sets even in large circuits in a reasonable time with a high fault coverage.

The benchmark set was defined in 1985 to compare different test generation systems [35]. The results on a Sun 3/260 are given in Table 1, which shows the name of the circuit, the number of gates, the size of the collapsed fault set, and the number of primary inputs and outputs. For each circuit,

Table 1
Benchmark results.

Name	Gates	Faults	In	Out	Red	Ab	%	#	Time
432	160	524	36	7	1	3	99.24	68	34.0
499	202	758	41	32	8	0	98.94	62	32.6
880	383	942	60	26	0	0	100	74	68.3
1355	546	1574	41	32	8	2	99.36	92	126.7
1908	880	1879	33	25	5	5	99.47	124	245.2
2670	1193	2747	233	140	97	41	94.98	105	433.2
3540	1669	3428	50	22	127	25	95.57	175	703.9
5315	2307	5350	178	123	59	22	98.49	141	819.3
6288	2406	7744	32	32	34	0	99.56	37	265.5
7552	3512	7550	207	108	88	122	97.22	281	2223.6

we show the number of redundant faults detected (Red), the number of aborted faults (Ab), for which the procedure did not find a test or could not detect redundancy, the fault coverage (%), and the number of test patterns (#) generated. Execution times are shown for test generation only (Time).

The program obtains quite high fault coverage for all test examples. The first test patterns detect many new faults and then the number decreases slowly. The same behavior can be observed for the other systems. This shows a *tradeoff* between fault coverage and execution time. By investing more time, a slightly better fault coverage can be obtained.

The average time needed to find one test pattern for each of the example circuits grows nearly linearly with the size of the circuit. This is to be expected since, with our program, the whole circuit must be simulated to find a test pattern.

Table 2 shows the results of several special-purpose systems. It is very hard to compare two different test generation algorithms in a fair way. Fault coverage can be compared relatively easily since most systems use the same fault set. Execution times vary widely. Systems are implemented

Table 2
Benchmark comparison.

Name	Socrates		FAN	D-Alg	AIDSTG	
	%	sec	%	%	%	sec
432	99.24	5.3	94.7	97.4	99.05	70
499	98.94	24.9	93.5	68.5	99.29	101
880	100	5.7	100	100	100	107
1355	99.49	34.3	93.5	58.2	99.64	301
1908	99.52	63.1	94.6	95.0	99.59	533
2670	95.49	61.1	93.2	95.3	96.25	809
3540	95.95	89.0	92.0	94.4	95.90	1398
5315	98.88	45.4	98.2	98.5	99.21	934
6288	99.56	32.8	98.5	99.1	99.48	892
7552	98.25	243.5	93.7	96.3	98.26	2121

on different machines in different languages. For some systems, only total time is given, for others only test generation time. However, we can observe two main points. First, the fault coverage of our approach is quite good, in some cases exceeding some of the specialized programs. This means that the model and the propagation mechanisms used are quite powerful, finding a test pattern even in difficult cases. Second the experimental results indicate that the performance of the program is within a constant factor of the best specialized algorithms. This is encouraging given the effort spent in the development of these hand-crafted programs, the specialized nature of the problem, and the room left for optimization in constraint languages. It shows that a general and flexible programming language like cc(FD), especially designed for short development time and rapid prototyping, enables us to design a small declarative program whose efficiency is within a constant factor of the best special-purpose algorithms.

4. The car sequencing problem

The second application we consider is the so-called car sequencing problem. This was motivated by an article published in *AI Expert* [48] which posed the problem as a challenge for AI technology. We describe a solution using cc(FD).

4.1. Problem statement

Cars in production are placed on an assembly line that moves through various production units responsible for installing such options as air-conditioning, radios, etc. The assembly line can be viewed as composed of slots, and each car must be allocated to a single slot. However, the cars cannot be allocated arbitrarily: the production units have limited capacity

9V		10V(M)	ATWIG	Brglez	FAN	
%	sec	%	%	%	%	sec
99.1	8.1	98.9	95.9	99.24	93.7	3.6
98.9	18.1	98.9	88.0	98.94	99.4	16.2
100	26.3	100	99.2	100	100	1.3
99.5	72.6	98.7	86.7	97.27	99.5	13.5
99.6	143.2	99.4	81.9	99.52	99.5	13.5
95.4	517	93.7	81.1	95.34	95.7	49.4
96.1	452	94.7	90.0	95.71	96.0	42.9
98.9	844	98.6	96.4	98.82	98.9	19.7
99.6	1039	69.9	99	99.56	99.5	31.7
98.1	1446	96.6	92.2	98.19	98.2	118.6

Table 3
A car sequencing example.

Classes	1	2	3	4	5	6	Capacity
Option 1	y	–	–	–	y	y	1/2
Option 2	–	–	y	y	–	y	2/3
Option 3	y	–	–	–	y	–	1/3
Option 4	y	y	–	y	–	–	2/5
Option 5	–	–	y	–	–	–	1/5
Cars	1	1	2	2	2	2	

Table 4
Car sequencing: a solution.

	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}
Class 1	+	–	–	–	–	–	–	–	–	–
Class 2	–	+	–	–	–	–	–	–	–	–
Class 3	–	–	–	+	–	–	–	–	+	–
Class 4	–	–	–	–	–	+	+	–	–	–
Class 5	–	–	–	–	+	–	–	+	–	–
Class 6	–	–	+	–	–	–	–	–	–	+

and they need time to set up the options on the cars as the assembly line is moving in front of the unit. These *capacity constraints* are formalized using constraints of the form *r outof s*, which indicate that the unit is able to produce at most *r* cars with the option out of each sequence of *s* cars. The car sequencing problem amounts to finding an assignment of cars to the slots that satisfies the capacity constraints.

We illustrate the problem on a simple example. In the example and the algorithm below, cars requiring the same set of options are clustered into classes, since they cannot be distinguished for any useful purpose in the algorithm. Table 3 presents a problem with five options, six classes, and ten cars. Here "y" means that a particular option is required by the class, "–" means that it is not required. The capacity constraint *r/s* should be read as *r outof s*. For example, two cars of class 6 need to be produced. They require options 1 and 2. The capacity unit for option 1 has a constraint "1 outof 2", indicating that no two consecutive cars can require the option since the unit cannot set up the option on the two consecutive cars while the line is moving.

The search space in this problem is made up by the possible values for the slots of the assembly line. Tables 4 and 5 depict a solution to the simple example, where "–" denotes an inconsistent value and "+" an assigned value; the assembly line itself is best described by the options selected for each slot.

Table 5
Car sequencing: the assembly line in a solution.

	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}
Option 1	+	−	+	−	+	−	−	+	−	+
Option 2	−	−	+	+	−	+	+	−	+	+
Option 3	+	−	−	−	+	−	−	+	−	−
Option 4	+	+	−	−	−	+	+	−	−	−
Option 5	−	−	−	+	−	−	−	−	+	−

4.2. Problem solution

As is typical of finite-domain programs, the program contains two parts: a constraint part that generates the problem constraints and a choice part that assigns values to some of the problem variables. In this section we describe the variables used in modeling the problem, the constraints expressed in terms of these variables as well as short programs describing how these constraints may be generated, and the way choices are performed. We then describe the basic program and show how to improve its efficiency.

Conventions. We assume that we are given n classes of cars. Each class i contains n_i cars ($n_i \geqslant 0$) such that the total numbers of cars is $ns = \sum_{i=1}^{n} n_i$. We also assume m different options. For each class i and option j, we have a Boolean o_{ij} which is true if class i requires option j and false otherwise. For convenience, we represent *true* by 1 and *false* by 0.

4.2.1. Problem variables

The first step towards the solution is to identify the problem variables in terms of which the constraints are stated. To each slot i ($1 \leqslant i \leqslant ns$), we associate a variable S_i denoting the class of cars assigned to the slot. These variables, called the *slot variables*, represent the main output of the program.

Each slot i is also associated with m variables, one for each option denoted $O_i^1, O_i^2, \ldots, O_i^m$. O_i^j ($1 \leqslant i \leqslant ns$ and $1 \leqslant j \leqslant m$) is equal to 1 if the class S_i (the class assigned to slot i) requires option j and 0 otherwise. These variables are called the *option variables*. There are $O(ns)$ slot variables and $O(ns \times m)$ option variables. In the above example, there are 10 slot variables $(S_1, \ldots, S_{10})$ and 50 option variables $O_1^1, \ldots, O_1^5, \ldots, O_{10}^1, \ldots, O_{10}^5$.

4.2.2. Domain constraints

We now turn to the problem constraints. The first constraints are the domain constraints for the slot and option variables. Each slot variable S_i has a constraint $S_i \in \{1, \ldots, n\}$ and each option variable O_i^j has a constraint $O_i^j \in \{0, 1\}$. In other words, each slot variable can be assigned a class of

cars while each slot variable is assigned a Boolean value. A simple recursive program can be used to generate these constraints:

```
state_domains([],Low,High).
state_domains([F|T],Low,High) :-
  F ∈ Low..High,
  state_domains(T,Low,High).
```

The goal `state_domains(L,0,1)` imposes a Boolean domain to all variables in the list `L`.

The domain constraints generated for the example in Table 3 are as follows:

$$S_1 \in \{1,\ldots,6\}, \ldots, S_{10} \in \{1,\ldots,6\},$$
$$O_1^1 \in \{0,1\},\ldots,O_1^5 \in \{0,1\}, \ldots, O_{10}^1 \in \{0,1\},\ldots,O_{10}^5 \in \{0,1\}.$$

4.2.3. Capacity constraints

The capacity constraints are stated in terms of the slot variables. If the capacity constraint for option j $(1 \leqslant j \leqslant m)$ is of the form r *outof* s, constraints must be generated of the form

$$O_i^j + \cdots + O_{i+s-1}^j \leqslant r, \quad 1 \leqslant i \leqslant ns - s + 1.$$

For instance, option 1 (1 outof 2) generates the constraints

$$O_1^1 + O_2^1 \leqslant 1,$$
$$\ldots$$
$$O_9^1 + O_{10}^1 \leqslant 1,$$

while option 2 (2 outof 3) generates the constraints

$$O_1^2 + O_2^2 + O_3^2 \leqslant 2,$$
$$O_2^2 + O_3^2 + O_4^2 \leqslant 2,$$
$$\ldots$$
$$O_8^2 + O_9^2 + O_{10}^2 \leqslant 2.$$

A program can be written to generate all constraints of the form "r outof s". Specialized to a constraint of the type "1 outof 2", it looks like

```
atmost1outof2([]).
atmost1outof2([O]).
atmost1outof2([O1,O2|Os]) :-
  O1 + O2 ⩽ 1,
  atmost1outof2([O2|Os]).
```

The above program generates linear inequalities for the variables. Overall there are $O(ns \times m)$ capacity constraints.

4.2.4. Demand constraints

It is also necessary to make sure that the cars requested are produced. For each class i $(1 \leqslant i \leqslant n)$, a constraint

$$exactly(n_i, [S_1, \ldots, S_{ns}], i)$$

has to be generated, where $S_1, \ldots, S_{ns}$ are the slot variables and n_i is the number of cars in class i. The constraint `exactly(N,L,M)` holds iff there are exactly `N` variables in the list `L` whose values are equal to `M`.

In fact, since there are ns slot variables and each of them will be assigned to a class (and thus a car), it is only necessary to make sure that the assignment produces no more cars from a class than are actually necessary. Hence the above constraints reduce to `atmost` constraints,

$$atmost(n_i, [S_1, \ldots, S_{ns}], i).$$

A constraint `atmost(N,L,M)` holds iff there are at most `N` variables in the list `L` whose values are equal to `M`.

To express the `atmost` constraint, we make use of the cardinality combinator. The idea is that a constraint

$$atmost(n_i, [S_1, \ldots, S_{ns}], i)$$

corresponds to the cardinality formula

$$\#(*, n_i, [S_1 = i, \ldots, S_{ns} = i]).$$

In other words, the cardinality formula makes sure that at most n_i constraints in $[S_1 = i, \ldots, S_{ns} = i]$ hold, and hence that at most n_i slots are assigned a car from class i. There are n demand constraints. The following constraints are generated for our example:

```
#(*,1,[S1 = 1,...,S10 = 1]),
...
#(*,2,[S1 = 6,...,S10 = 6]).
```

These cardinality formulas can be generated in a simple way by the following program which, given a list `L` and two integers `N` and `M`, makes sure that at most `N` elements of the list `L` are assigned to `M`.

```
atmost(N,L,M) :-
  collect_equalities(L,M,Eqs),
  #(*,N,Eqs).

collect_equalities([],M,[]).
collect_equalities([F|T],M,[F = M | Eqs]) :-
  collect_equalities(T,M,Eqs).
```

The first goal in the `atmost` predicate collects equalities between the value `M` and the elements of the list `L`, while the cardinality combinator makes sure that at most `N` of them are true.

4.2.5. Link constraints

Although all constraints seem to have been enforced at this point, an important step is still missing. The option variables and slot variables have been left completely unconnected so that a slot variable can be assigned a value without influencing its corresponding option variables and vice versa. To ensure correctness and to perform effective pruning, it is necessary to link the slot and option variables. The link is achieved by generating constraints of the form `element(I,L,V)` which hold iff element `I` of the list `L` is equal to `V`. Each option j will be connected with slot i by the constraint

$$element(S_i, [o_{1j}, \ldots, o_{nj}], O_i^j),$$

where $o_{1j}, \ldots, o_{nj}$ are the 0–1 values specifying which classes require option j. In the example, the connection between the slots and options is enforced by the constraints

```
element(S1,[1,0,0,0,1,1],O1^1),
...
element(S1,[0,0,1,0,0,0],O1^5),
...
element(S10,[1,0,0,0,1,1],O10^1),
...
element(S10,[0,0,1,0,0,0],O10^5).
```

There are $O(ns \times m)$ relation constraints.

How should a constraint `element(I,L,V)` be defined? Obviously, it is desirable that, as soon as `I` is given a value, `V` is assigned its corresponding value (for instance, in the above first constraint, if S_1 is assigned to 3, O_1^1 must be assigned to 0). On the other hand, much more pruning can be achieved. In particular, as soon as S_1 is restricted to the values 1, 4, and 5, O_1^1 must be given the value 1. In the same way, as soon as O_1^1 is assigned the value 1, S_1 is restricted to take values in {1,4,5}. In other words, we would like `element(I,L,V)` to be *arc consistent.*

To enforce arc-consistency on `element(I,L,V)`, it is sufficient to generate cardinality constraints of the form

$$V = e \Leftrightarrow I \in \{i_1, \ldots, i_p\}$$

where `e` is a value in `L` and $i_1, \ldots, i_p$ are all the positions in list `L` whose value is `e`. In general, a constraint must be generated for each value in `L`, although this is not necessary in the car sequencing application (since only

Boolean values are used). For instance, the first `element` constraint of our example generates the constraint

$$O_1^1 = 1 \Leftrightarrow S_1 \in \{1,5,6\}.$$

A simple program can be written to generate the above constraints. The equivalence $\Leftrightarrow$ should be understood as an abbreviation for a cardinality formula and illustrates the fact that the cardinality operator can be used to enforce arc-consistency of any constraint (preserving the complexity bounds of the optimal algorithm of [44]).

4.2.6. Basic program

We now present the basic program, which amounts to stating the constraints and generating values for the slots variables.

```
sequencing(Line,InstanceData) ←
  state_constraints(Line,InstanceData),
  generate_values(Line).

state_constraints(Line,
                  [NbSlots,NbOptions,NbClasses,
                   OptionInfo,CarInfo]):-
  generate_slots(Line,NbSlots),
  generate_option_variables(Options,NbSlots,NbOptions),
  state_domain_constraints(Line,1,NbClasses),
  state_domain_constraints(Options,0,1),
  state_demand_constraints(Line,CarInfo),
  state_capacity_constraints(Options,OptionInfo),
  state_link_constraints(Line,Options,OptionInfo).
```

The arguments of the predicate are the list of slots variables and the data characterizing the instance. The generation of constraints creates as many variables as there are slots in the assembly line, creates the option variables, and states all the above-mentioned constraints. Generating the constraints can be done by simple recursive programs (as has been shown in the above presentation) and poses no particular difficulty. Assigning a value to the slot variables produces a solution satisfying the constraints. The generation of values simply assigns to each of the slot variables a value between 1 and n, that is, a class of cars.

4.2.7. Improving efficiency

The above program provides us with a reasonably efficient solution to the car sequencing problem. It is possible, however, to speed up the program significantly by exploiting properties of the solutions and making choices wisely.

Redundant (surrogate) constraints

A traditional technique in operations research amounts to generating *surrogate constraints*: constraints which are not strictly necessary to guarantee correctness of the application but perform pruning by exploiting properties that must be satisfied by the solutions. In other words, the constraints are redundant semantically but not operationally.

The car sequencing problem has a surrogate constraint worth exploiting. Assume that option j has a capacity constraint r outof s. We know that the last s slots contain only r cars, so the other slots must contain all the remaining cars having that option. If p cars require option j, we can generate a constraint

$$O_1^j + \cdots + O_{ns-s}^j \geqslant p - r.$$

More generally, the last $k \times s$ $(k = 1, 2, \ldots, ns/s)$ slots can contain only $k \times r$ cars and hence the constraints

$$O_1^j + \cdots + O_{ns-k\times s}^j \geqslant p - k \times r$$

can be generated.

In our example, for instance, option 1 is requested by five cars and has capacity "1 outof 2". Since only one car can be scheduled in the last two slots, four cars must be sequenced in the first eight slots. Pursuing the reasoning, we can generate the following constraints:

$$O_1^1 + \cdots + O_8^1 \geqslant 4,$$
$$O_1^1 + \cdots + O_6^1 \geqslant 3,$$
$$O_1^1 + \cdots + O_4^1 \geqslant 2.$$

The effect of these constraints is to prune the search space early and to escape deep backtracking and thrashing by recognizing and avoiding failures as soon as possible. It is not difficult to write a recursive program generating the above constraints.

First-fail principle

Following [34], we make use of the first-fail principle in the choice process: that is we try to choose the most constrained variable to be instantiated next. In the car sequencing, this is done by choosing the variable with the smallest domain (i.e. the one that can be given the smallest number of values) and, in case of equality, the more demanding one in terms of the options.

Table 6
Car sequencing: search space after one choice.

	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}
Class 1	+	–	–	–	–	–	–	–	–	–
Class 2	–									
Class 3	–									
Class 4	–									
Class 5	–	–	–							
Class 6	–	–								

Table 7
Car sequencing: the assembly line after one choice.

	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}
Option 1	+	–								
Option 2	–									
Option 3	+	–	–							
Option 4	+									
Option 5										

4.3. Example

The above program first states the constraints and then makes choices. After the first choice (i.e. $S_1 = 1$), the search space and the assembly line are depicted in Tables 6 and 7. Now, giving to S_2 its first possible value, i.e. 2, leads directly to the solution presented at the beginning of the example. Indeed, this choice immediately removes the value 2 for all other slot variables and prevents variables S_3, S_4, and S_5 from taking the value 4 because of option 4. This intermediate state is depicted in Tables 8 and 9.

But the surrogate constraints for option 2 require that S_3 and S_4 be assigned a class including option 2, since six cars with option 2 must be produced. The effect of these assignments is to fix all option variables concerning option 2 and to remove possible values from the slot variables. The search space and assembly line at that stage are depicted in Tables 10 and 11.

At this point, the demand constraints for class 5 come into play. Two cars of class 5 must be produced; since only two places are left for them, they

Table 8
Car sequencing: search space after two choices (part I).

	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}
Class 1	+	–	–	–	–	–	–	–	–	–
Class 2	–	+	–	–	–	–	–	–	–	–
Class 3	–	–								
Class 4	–	–	–	–	–					
Class 5	–	–	–							
Class 6	–	–								

Table 9
Car sequencing: the assembly line after two choices (part I).

	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}
Option 1	+	−								
Option 2	−	−								
Option 3	+	−	−							
Option 4	+	+	−	−	−					
Option 5		−								

Table 10
Car sequencing: search space after two choices (part II).

	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}
Class 1	+	−	−	−	−	−	−	−	−	−
Class 2	−	+	−	−	−	−	−	−	−	−
Class 3	−	−								
Class 4	−	−	−	−	−					
Class 5	−	−	−	−		−	−		−	−
Class 6	−	−								

are assigned immediately. This leads to the search space and assembly line depicted in Tables 12 and 13. The final step amounts to using the surrogate constraints for option 1. These constraints fix all options concerning option 1 and lead to the solution depicted earlier in this paper. Note that here a solution was found in two choices without any backtracking.

Table 11
Car sequencing: the assembly line after two choices (part II).

	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}
Option 1	+	−								
Option 2	−	−	+	+	−	+	+	−	+	+
Option 3	+	−	−							
Option 4	+	+	−	−	−					
Option 5		−								

Table 12
Car sequencing: search space after two choices (part III).

	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}
Class 1	+	−	−	−	−	−	−	−	−	−
Class 2	−	+	−	−	−	−	−	−	−	−
Class 3	−	−			−			−		
Class 4	−	−	−	−	−			−		
Class 5	−	−	−	−	+	−	−	+	−	−
Class 6	−	−			−			−		

Table 13
Car sequencing: the assembly line after two choices (part III).

	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}
Option 1	+	−		−	+	−	−	+		
Option 2	−	−	+	+	−	+	+	−	+	+
Option 3	+	−	−		+			+		
Option 4	+	+	−	−	−			−		
Option 5		−			−			−		

4.4. Computation results

A large number of experiments have been run to evaluate the efficiency of the algorithm on various problem instances. Remember that, since the problem is NP-complete, it is always possible to construct an instance that will require exponential time (whatever the proposed program).

The basic assumption in our experiments was that the assembly line supported five different options with the following capacity constraints: 1 outof 2, 2 outof 3, 1 outof 3, 2 outof 5, and 1 outof 5. Given this assumption, several parameters were still left free: the number of cars *ns*, the particular requirements for the cars, and the utilization of each production unit. In our experiments, *ns* varied from 5 to 200 and random data were generated for the utilization percentage and the options required by the cars. This random generation guarantees an overall percentage of utilization of the resources. Typically, we would ask for 70% or 80% but experiments have shown that it is possible to make this percentage even higher. For each experiment, large data samples (around 100) were generated.

When $ns < 50$, the program finds a solution in a few seconds, generally with very little if any backtracking. When $ns = 50$, the scheduling time is around 15 seconds on a Sun 3/160. Once again, little backtracking was needed to reach a solution.

When $ns = 100$, the average scheduling time is less than a minute with

Table 14
100 cars sequencing with about 70% of option utilization.

ns	*n*	%-1	%-2	%-3	%-4	%-5	CPU time
100	24	72	72	72	67	70	52 sec.
100	24	74	75	100	90	60	58 sec.
100	21	84	68	75	60	75	56 sec.

Table 15
100 cars sequencing with about 80% of option utilization.

ns	*n*	%-1	%-2	%-3	%-4	%-5	CPU time
100	25	88	84	72	77	75	62 sec.
100	22	78	80	84	90	70	58 sec.
100	21	80	81	75	72	75	59 sec.

Table 16
100 cars sequencing with about 70% of option utilization.

ns	*n*	%-1	%-2	%-3	%-4	%-5	CPU time
200	29	86	77	89	85	85	336 sec.
200	29	89	82	83	83	95	340 sec.
200	31	84	81	95	82	100	345 sec.

a utilization percentage around 70%. Table 14 reports some typical results. The first column *ns* is the number of cars, the second column *n* indicates the number of different classes, the next five columns indicate the utilization percentage of each option and the last column shows the CPU time required to generate the constraints and find a schedule. Increasing the utilization percentage to 80% increases the CPU time by only a few seconds. Typical results are shown in Table 15.

When $ns = 200$, the average scheduling time is around 5 minutes for an overall utilization percentage of 70% and thus scheduling time does not change when we increase the percentage to 80%. Table 16 reports some typical results.

Note that the potential search space to explore in the last example is 200^{31}. The program must generate 200 slots variables, 1000 option variables, more than 1000 cardinality constraints, and about 3000 numerical constraints. Only the slot variables, however, need to be instantiated to give a solution however, whereas an integer programming solution would require more than 7000 variables, of which 6000 would need to be instantiated to find a solution.

In the experiments, the execution time was found to increase quadratically in the average. No instance was found that could not be solved (even when $ns = 400$) although such instance could be constructed since the problem is NP-complete.

5. Conclusion

We have shown how to solve two practical combinatorial search problems using cc(FD), a successor to CHIP using consistency techniques on finite domains. The test generation problem is a well-known problem in digital circuit design and we have presented an original and complete algorithm for the task based on constraint satisfaction. The car sequencing problem was posed as a challenge for AI technology and a constraint-based solution has also been presented.

Both problems can be expressed concisely and declaratively in cc(FD) and require a small fraction of the development necessary to obtain "equivalent" procedural programs. The resulting programs can be easily extended,

modified, and specialized due to the declarative nature of the language. In addition, the resulting algorithms can solve large instances of the problems in reasonable time and are competitive with procedural programs to a constant factor.

Current and future research is devoted to (1) design aspects in order to capture more abstractions useful in combinatorial search problems and (2) to implementation issues (to reduce the constant factor with respect to procedural languages).

Appendix A. Formalization of the semantics of cc(FD)

Here we formalize, following [54], the operational semantics of cc(FD) using a structural operational semantics [50]. Those interested in a broader and more rigorous handling of the semantics can refer to [36] and [55,56]; [36] contains a complete description of the CLP scheme while [55,56] respectively describe the operational and denotational semantics (in terms of information systems and closure operators) of the cc framework.

A.1. The CLP scheme

The operational semantics makes use of a transition system.

Definition A.1. A *transition system* is a triple $\langle \Gamma, T, \longmapsto \rangle$ where Γ is a set of configurations, $T \subseteq \Gamma$ is the set of terminal configurations and $\longmapsto \subseteq \Gamma \times \Gamma$ is the transition relation satisfying

$$\forall \gamma \in T, \quad \forall \gamma' \in \Gamma, \quad \gamma \not\longmapsto \gamma'.$$

The configurations of the transition system are the computation states $\langle G \square \sigma \rangle$. When the goal part is empty, we represent the configuration by the constraint part only; when the constraint part is empty, we represent the configuration by the goal part only. Terminal configurations are simply successful computation states (i.e. constraint stores) or the terminal `block` to denote blocking.

A transition $\gamma \longmapsto \gamma'$ can be read as "configuration γ nondeterministically reduces to γ'". The transition rules in this paper are presented using the format

$$\frac{\langle \text{ condition } 1 \rangle \quad \ldots \quad \langle \text{condition } n \rangle}{\gamma \longmapsto \gamma'}$$

expressing the fact that a transition from γ to γ' can take place if the conditions are fulfilled. We are now ready to present the various transition rules.

Goal Reduction. A goal can be reduced to the body of a clause if the constraint store is consistent with the equality constraints.

$$\frac{\begin{array}{l} p(s_1,\ldots,s_n) \ :- \ B_1,\ldots,B_m \ \in P \\ \mathcal{C} \models (\exists)\ (\sigma \wedge t_1 = s_1 \wedge \cdots \wedge t_n = s_n) \end{array}}{\begin{array}{l} \langle\, p(t_1,\ldots,t_n) \ \square \ \sigma \,\rangle \\ \quad \longmapsto \langle\, B_1,\ldots,B_m \ \square \ \sigma \wedge t_1 = s_1 \ \& \ \cdots \ \& \ t_n = s_n \,\rangle \end{array}}$$

In the above transition rule, $p(t_1,\ldots,t_n)$ is the atom selected, P denotes the program, $(\exists)\ (\psi)$ represents the existential closure of ψ, and the program clause has been renamed properly to avoid sharing any variable with the goal.[4] The rule expresses formally the first kind of computation step described in the informal presentation. If there exists a clause in the program with the same predicate name as the selected atom (condition 1) and if the equality constraints are consistent with the constraint store (condition 2), then a computation step is possible. The new computation state is obtained from the old one by replacing the selected atom by the clause body and adding the equality constraints to the constraint store.

Constraint Solving. A constraint can be removed from the goal part iff it is consistent with the constraint store.

$$\frac{\mathcal{C} \models (\exists)\ (\sigma \wedge c)}{\langle c \ \square \ \sigma \,\rangle \longmapsto \sigma \ \& \ c}$$

This rules captures what was informally described by the second type of computation step.

Conjunction. If any of the goals in a conjunction can make a transition, the whole conjunction can make a transition as well and the constraint store is updated accordingly. This is the traditional interleaving rule.

$$\frac{\langle\, G_1 \ \square \ \sigma \,\rangle \longmapsto \langle\, G_1' \ \square \ \sigma' \,\rangle}{\begin{array}{l} \langle\, G_1, G_2 \ \square \ \sigma \,\rangle \longmapsto \langle\, G_1', G_2 \ \square \ \sigma' \,\rangle \\ \langle\, G_2, G_1 \ \square \ \sigma \,\rangle \longmapsto \langle\, G_2, G_1' \ \square \ \sigma' \,\rangle \end{array}}$$

[4] In recent work [56], Saraswat et al. give an operational semantics precluding the need for renaming.

$$\frac{\langle G_1 \;\square\; \sigma \rangle \longmapsto \sigma'}{\begin{array}{l}\langle G_1, G_2 \;\square\; \sigma \rangle \longmapsto \langle G_2 \;\square\; \sigma' \rangle \\ \langle G_2, G_1 \;\square\; \sigma \rangle \longmapsto \langle G_2 \;\square\; \sigma' \rangle\end{array}}$$

These are the only rules necessary for the CLP scheme.

A.2. The implication combinator

We now describe precisely the semantics of the implication combinator.

Implication. An implication $c \rightarrow A$ never fails. If c is entailed by the constraint store, it reduces to the body A. If $\neg c$ is entailed by the constraint store, the implication terminates successfully. Otherwise, the implication blocks.

$$\frac{\mathcal{C} \models (\forall)\ (\sigma \rightarrow c)}{\langle c \rightarrow A \;\square\; \sigma \rangle \longmapsto \langle A, \sigma \rangle}$$

$$\frac{\mathcal{C} \models (\forall)\ (\sigma \rightarrow \neg c)}{\langle c \rightarrow A \;\square\; \sigma \rangle \longmapsto \sigma}$$

$$\frac{\begin{array}{l}\mathcal{C} \models \neg(\forall)\ (\sigma \rightarrow c) \\ \mathcal{C} \models \neg(\forall)\ (\sigma \rightarrow \neg c)\end{array}}{\langle c \rightarrow A \;\square\; \sigma \rangle \longmapsto \mathit{block}}$$

A.3. The cardinality combinator

The precise behavior of the combinator can be described by the following transition rules taken from [73].

Trivial Satisfaction. If $l \leqslant 0$ and u is greater than or equal to the number of constraints $c_1, \ldots, c_n$, then $\#(l, u, [c_1, \ldots, c_n])$ is trivially satisfied:

$$\frac{l \leqslant 0 \wedge n \leqslant u}{\langle \#(l, u, [c_1, \ldots, c_n]) \;\square\; \sigma \rangle \longmapsto \sigma}$$

Positive Satisfaction. A formula $\#(n, u, [c_1, \ldots, c_n])$ with $n \leqslant u$ can be satisfied only if the conjunction $c_1 \wedge \cdots \wedge c_n$ is consistent with the constraint store:

$$\frac{\begin{array}{l} l \leqslant u \wedge l = n \\ \mathcal{C} \models (\exists)\,(\sigma \wedge c_1 \wedge \ldots \wedge c_n) \end{array}}{\langle\, \#(l, u, [c_1, \ldots, c_n])\ \square\ \sigma \,\rangle \longmapsto \sigma\ \&\ c_1\ \&\ \ldots\ \&\ c_n}$$

Negative Satisfaction. A formula $\#(l, 0, [c_1, \ldots, c_n])$ with $l \leqslant 0$ can be satisfied only if the conjunction $\neg c_1 \wedge \cdots \wedge \neg c_n$ is consistent with the constraint store:

$$\frac{\begin{array}{l} l \leqslant u \wedge u = 0 \\ \mathcal{C} \models (\exists)\,(\sigma \wedge \neg c_1 \wedge \ldots \wedge \neg c_n) \end{array}}{\langle\, \#(l, u, [c_1, \ldots, c_n])\ \square\ \sigma \,\rangle \longmapsto \sigma\ \&\ \neg c_1\ \&\ \cdots\ \&\ \neg c_n}$$

The above three rules make up the basic cases for the cardinality combinator. Two of them allow the inference of primitive constraints, and hence prune the search space with the help of the transition rules for conjunction.

Positive Reduction. When a constraint c_i is entailed by the constraint store, the cardinality formula can be simplified by dropping the constraint and decrementing the bounds.

$$\frac{\begin{array}{l} \mathcal{C} \models (\forall)(\sigma \rightarrow c_i) \\ 0 < l < n \wedge l \leqslant u \ \vee\ 0 < u < n \wedge l \leqslant 0 \end{array}}{\begin{array}{l} \langle\, \#(l, u, [c_1, \ldots, c_i, \ldots, c_n])\ \square\ \sigma \,\rangle \\ \quad \longmapsto \langle\, \#(l-1, u-1, [c_1, \ldots, c_{i-1}, c_{i+1}, \ldots, c_n])\ \square\ \sigma \,\rangle \end{array}}$$

The condition on l and u forces the rule to be mutually exclusive with the three satisfaction rules.

Negative Reduction. When the negation of a constraint c_i is entailed by the constraint store (i.e. c_i inconsistent with σ), the cardinality formula can be simplified by dropping the constraint:

$$\frac{\begin{array}{l} \mathcal{C} \models (\forall)(\sigma \rightarrow \neg c_i) \\ 0 < l < n \wedge l \leqslant u \ \vee\ 0 < u < n \wedge l \leqslant 0 \end{array}}{\begin{array}{l} \langle\, \#(l, u, [c_1, \ldots, c_i, \ldots, c_n])\ \square\ \sigma \,\rangle \\ \quad \longmapsto \langle\, \#(l, u, [c_1, \ldots, c_{i-1}, c_{i+1}, \ldots, c_n])\ \square\ \sigma \,\rangle \end{array}}$$

The above two rules achieve progress towards the satisfaction rules by reducing the number of constraints and (possibly) the bounds. But the computation with the cardinality combinator may now block as none of the constraints can be decided upon (for entailment) with respect to the constraint store.

Blocking. The cardinality combinator blocks if there is no constraint c_i such that either c_i or its negation $\neg c_i$ is entailed by the constraint store and none of the satisfaction rules apply:

$$\frac{\begin{array}{l} \mathcal{C} \not\models (\forall)(\sigma \rightarrow c_j) \quad \text{for all } 1 \leqslant j \leqslant n \\ \mathcal{C} \not\models (\forall)(\sigma \rightarrow \neg c_j) \quad \text{for all } 1 \leqslant j \leqslant n \\ 0 < l < n \wedge l \leqslant u \ \vee \ 0 < u < n \wedge l \leqslant 0 \end{array}}{\langle \#(l, u, [c_1, \ldots, c_n]) \ \square \ \sigma \rangle \longmapsto \mathit{block}}$$

We now reconsider our simple example and indicate the transition rules used in the derivation.

```
⟨#(1,2,[X=4, Y=10]) & X>6 □ ε⟩
    ↓ (conjunction)
⟨#(1,2,[X=4, Y=10]) □ X>6⟩
    ↓ (negative reduction)
⟨#(1,2,[Y=10]) □ X>6⟩
    ↓ (positive satisfaction)
X>6 & Y=10
```

A.4. Operational semantics

The actual operational semantics of the language can be defined in terms of its success, divergence, and failure sets. We use the notation $P \vdash$ to denote the fact that the transition occurs in the context of program P. We denote by $\overset{*}{\longmapsto}$ the transitive closure of $\longmapsto$ and say that a configuration γ diverges in program P if there exists an infinite sequence of transitions

$$P \vdash \gamma \longmapsto \gamma_1 \longmapsto \cdots \longmapsto \gamma_i \longmapsto \cdots.$$

The operational semantics is now given in terms of three sets: the success, divergence, and blocking sets:

$$\begin{aligned} SS[P] &= \{G \mid P \vdash G \overset{*}{\longmapsto} \sigma\}, \\ DS[P] &= \{G \mid G \text{ diverges in } P\}, \\ BS[P] &= \{G \mid P \vdash G \overset{*}{\longmapsto} \mathit{block}\}. \end{aligned}$$

The failure set can now be defined in terms of the above three sets:

$$FS[P] = \{G \mid G \notin SS[P] \cup DS[P] \cup BS[P]\}.$$

Another semantic definition can be given to capture the results of the computation:

$$RES[P, G] = \{\sigma \mid P \vdash G \overset{*}{\longmapsto} \sigma\}$$

In order to achieve the above semantics, the CLP language should be embedded with a complete constraint solver; this means that, given a constraint σ, the constraint solver should return *true* if $\mathcal{C} \models (\exists)(\sigma)$ and *false* otherwise.

Acknowledgement

The initial solutions of the applications were proposed while the authors were at ECRC (Munich). Conversations with Yves Deville and Vijay Saraswat significantly improved our presentation. We also thank the three reviewers for their careful comments and suggestions. In particular, the comments (and humor) of the second reviewer (who will recognize herself or himself easily) are (greatly) appreciated. Trina Avery helped correcting our English. This research was supported in part by the National Science Foundation under grant number CCR-9108032 and by the Office of Naval Research and the Defense Advanced Research Projects Agency under Contract N00014-91-J-4052.

References

[1] J.A. Abraham, Fault modeling in VLSI, in: T.W. Williams, ed., *VLSI Testing*, Advances in CAD for VLSI **5** (North-Holland, Amsterdam, 1986) 1–27, Chapter 1.

[2] A. Borning, The programming language aspects of ThingLab, a constraint-oriented simulation laboratory, *ACM Trans. Programm. Lang. Syst.* **3** (4) (1981) 353–387.

[3] P.S. Bottorff, Test generation and fault simulation, in: T.W. Williams, ed., *VLSI Testing*, Advances in CAD for VLSI **5** (North-Holland, Amsterdam, 1986) 29–64, Chapter 2.

[4] F. Brglez, P. Pownall and R. Hum, Applications of testability analysis: from ATPG to critical delay path tracing, in: *Proceedings IEEE International Test Conference* (1984).

[5] F. Brglez, P. Pownall and R. Hum, Accelerated ATPG and fault grading via testability analysis, in: *Proceedings IEEE International Symposium on Circuits and Systems*, Kyoto, Japan (1985) 695–698.

[6] W. Buttner and H. Simonis, Embedding Boolean expressions into logic programming, *J. Symbol. Comput.* **4** (1987) 191–205.

[7] J. Carlier and E. Pinson, Une methode arborescente pour optimiser la durée d'un JOB-SHOP, Tech. Rept. ISSN 0294-2755, I.M.A. (1986).

[8] W.T. Cheng, The back algorithm for sequential test-generation, in: *Proceedings IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD88)*, Rye Brook, NY (1988).

[9] N. Christofides, *Graph Theory: An Algorithmic Approach* (Academic Press, New York, 1975).

[10] K.L. Clark and F. McCabe, The control facilities of IC-PROLOG, in: D. Michie, ed., *Expert Systems in the Microelectronic Age* (Edinburgh University Press, Edinburgh, 1979) 122–149.

[11] W.F. Clocksin, Logic programming and digital circuit analysis, *J. Logic Programm.* **4** (1) (1987) 59–82.
[12] A. Colmerauer, An introduction to Prolog III, *Commun. ACM* **28** (4) (1990) 412–418.
[13] A. Colmerauer, H. Kanoui and M. Van Caneghem, Prolog, bases théoriques et developpements actuels, *T.S.I. (Techniques et Sciences Informatiques)* **2** (4) (1983) 271–311.
[14] T. Dean and M. Boddy, An analysis of time-dependent planning, in: *Proceedings AAAI-88*, St. Paul, MN (1988) 49–54.
[15] J. de Kleer, An assumption-based TMS, *Artif. Intell.* **28** (1986) 127–162.
[16] Y. Deville and P. Van Hentenryck, An efficient arc consistency algorithm for a class of CSP problems, in: *Proceedings IJCAI-91*, Sidney, Australia (1991).
[17] M. Dincbas and J.-P. Lepape, Metacontrol of logic programs in METALOG, in: *Proceedings International Conference on Fifth Generation Computer Systems (FGCS'84)*, Tokyo, Japan (1984) 361–370.
[18] M. Dincbas, H. Simonis and P. Van Hentenryck, Solving the car sequencing problem in constraint logic programming, in: *Proceedings European Conference on Artificial Intelligence (ECAI-88)*, Munich, Germany (1988).
[19] M. Dincbas, H. Simonis and P. Van Hentenryck, Solving large combinatorial problems in logic programming, *J. Logic Programm.* **8** (1–2) (1990) 75–93.
[20] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf and F. Berthier, The constraint logic programming language CHIP, in: *Proceedings International Conference on Fifth Generation Computer Systems*, Tokyo, Japan (1988).
[21] J. Doyle, A truth maintenance system, *Artif. Intell.* **12** (1979) 231–272.
[22] K. Eshghi, Application of meta-level programming to fault finding in logic circuits, in: *Logic Programming and Its Applications* (Ablex, Norwood, NJ, 1985) 208–219.
[23] R.E. Fikes. A heuristic program for solving problems stated as non-deterministic procedures, Ph.D. Thesis, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA (1968).
[24] M.S. Fox, Constraint-directed search: a case study of job-shop scheduling, Tech. Rept. CMU-CS-83-161, Carnegie-Mellon University, Pittsburgh, PA (1983).
[25] H. Fujiwara, FAN: a fanout oriented test pattern generation algorithm, in: *Proceedings. IEEE International Symposium on Circuits and Systems*, Kyoto, Japan (1985) 671–674.
[26] H. Fujiwara and T. Shimono, On the acceleration of test generation algorithms, *IEEE Trans. Comput.* **32** (1983) 1137–1144.
[27] H. Gallaire, Logic programming: further developments, in: *Proceedings IEEE Symposium on Logic Programming*, Boston, MA (1985) 88–99 (Invited Paper).
[28] H. Gallaire and C. Lasserre, Metalevel control for logic programs, in: *Logic Programming* (Academic Press, New York, 1982) 173–185.
[29] P. Goel, An implicit enumeration algorithm to generate tests for combinational logic circuits, *IEEE Trans. Comput.* **30** (1981) 215–222.
[30] T. Graf, Extending constraint handling in logic programming to rational arithmetic, Internal Report, ECRC, Munich, Germany (1987).
[31] T. Graf, P. Van Hentenryck, C. Pradelles and L. Zimmer, Simulation of hybrid circuits in constraint logic programming, *Comput. Math. Appl.* **20** (9–10) (1990) 45–56; Preliminary version in: *Proceedings IJCAI-89*, Detroit, MI (1989).
[32] E. Gullichsen, Heuristic circuit simulation using PROLOG, *Integr. VLSI J.* **3** (1985) 283–318.
[33] R. Gupta, Test-pattern generation for VLSI circuits in a Prolog environment, in: *Proceedings Third International Conference on Logic Programming*, London (1986) 528–535.
[34] R.M. Haralick and G.L. Elliot, Increasing tree search efficiency for constraint satisfaction problems, *Artif. Intell.* **14** (1980) 263–313.
[35] ISCAS, Special Session on ATPG, in: *Proceedings IEEE International Symposium on Circuits and Systems*, Kyoto, Japan (1985) 663–698.
[36] J. Jaffar and J.-L. Lassez, Constraint logic programming, in: *Proceedings 14th ACM Symposium on Principles of Programming Languages (POPL-87)*, Munich, Germany

(1987).
[37] J. Jaffar and S. Michaylov, Methodology and implementation of a CLP system, in: *Proceedings Fourth International Conference on Logic Programming*, Melbourne, Australia (1987).
[38] M. Kawai, K. Oozeki, M. Takahashi, M. Ono, Y. Ishizaka and T. Masui, Automatic test pattern generator for large combinational circuits, in: *Proceedings IEEE International Symposium on Circuits and Systems*, Kyoto, Japan (1985) 663–666.
[39] M. Kubale and D. Jackowski, A generalized implicit enumeration algorithm for graph coloring, *Commun. ACM* **28** (4) (1985) 412–418.
[40] J.-L. Lauriere, A language and a program for stating and solving combinatorial problems, *Artif. Intell.* **10** (1) (1978) 29–127.
[41] A. Lioy, Adaptive backtrace and dynamic partitioning enhance ATPG, in: *Proceedings IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD88)*, Rye Brook, NY (1988).
[42] A.K. Mackworth, Consistency in networks of relations, *Artif. Intell.* **8** (1) (1977) 99–118.
[43] M.J. Maher, Logic semantics for a class of committed-choice programs, in: *Proceedings Fourth International Conference on Logic Programming*, Melbourne, Australia (1987) 858–876.
[44] R. Mohr and T.C. Henderson, Arc and path consistency revisited, *Artif. Intell.* **28** (1986) 225–233.
[45] U. Montanari, Networks of constraints: fundamental properties and applications to picture processing, *Inf. Sci.* **7** (2) (1974) 95–132.
[46] M. Muarkami and H. Kikuchihara, Test generation for LSI circuits using extended nine-valued method, in: *Proceedings IEEE International Symposium on Circuits and Systems*, Kyoto, Japan (1985) 675–678.
[47] L. Naish, Negation and control in Prolog, Ph.D. Thesis, University of Melbourne, Australia (1985).
[48] B.D. Parrello, CAR WARS: the (almost) birth of an expert system, *AI Expert* **3** (1) (1988) 60–64.
[49] B.D. Parrello, W.C. Kabat and L. Wos, Job-shop scheduling using automated reasoning: a case study of the car-sequencing problem, *J. Autom. Reasoning* **2** (1) (1986) 1–42.
[50] G.D. Plotkin, A structural approach to operational semantics, Tech. Rept. DAIMI FN-19, CS Department, University of Aarhus, Denmark (1981).
[51] D.K. Pradhan, *Fault Tolerant Computing* (Prentice Hall, Englewood Cliffs, NJ, 1986).
[52] B.C. Rosales and P. Goel, Results from application of a commercial ATG system to large-scale combinatorial circuits, in: *Proceedings IEEE International Symposium on Circuits and Systems*, Kyoto, Japan (1985) 667–670.
[53] J. Roth, Diagnosis of automata failure: a calculus and a method, *IBM J. Res. Dev.* **10** (1966) 278–291.
[54] V.A. Saraswat, Concurrent constraint programming languages, Ph.D. Thesis, Carnegie-Mellon University, Pittsburgh, PA (1989).
[55] V.A. Saraswat and M. Rinard, Concurrent constraint programming, in: *Proceedings 17th ACM Symposium on Principles of Programming Languages*, San Francisco, CA (1990).
[56] V.A. Saraswat, M. Rinard and P. Panangaden, Semantic foundations of concurrent constraint programming, in: *Proceedings 19th ACM Symposium on Principles of Programming Languages*, Orlando, FL (1991).
[57] M. Schulz, E. Trischler and T. Sarfert, Socrates: a highly efficient automatic test pattern generation system, in: *Proceedings International Test Conference*, Washington, DC (1987).
[58] E. Shapiro, The family of concurrent logic programming languages, *Comput. Surv.* **21** (3) (1990) 413–510.
[59] H. Simonis, Test generation using logic programming, Tech. Rept. TR-LP-34, ECRC, Munich, Germany (1988).
[60] H. Simonis, Test generation using the constraint logic programming language CHIP, in: *Proceedings 6th International Conference on Logic Programming*, Lisbon, Portugal (1989).

[61] H. Simonis, ATPG revisited, Tech. Rept. TR-LP-56, ECRC, Munich, Germany (1990).
[62] H. Simonis and M. Dincbas, Using an extended Prolog for digital circuit design, in: *Proceedings IEEE International Workshop on AI Applications to CAD Systems for Electronics*, Munich, Germany (1987) 165–188.
[63] H. Simonis and M. Dincbas, Using logic programming for fault diagnosis in digital circuits, in: *Proceedings German Workshop on Artificial Intelligence (GWAI-87)*, Geseke, Germany (1987) 139–148.
[64] H. Simonis and T. Graf, Technology mapping in CHIP, Tech. Rept. TR-LP-44, ECRC, Munich, Germany (1990).
[65] H. Simonis, H.N. Nguyen and M. Dincbas, Verification of digital circuits using CHIP, in: G.J. Milne, ed., *Proceedings IFIP WG 10.2 International Working Conference on the Fusion of Hardware Design and Verification*, Glasgow, Scotland (1988).
[66] G.J. Sussman and G.L. Steele Jr, CONSTRAINTS—a language for expressing almost-hierarchical descriptions, *Artif. Intell.* **14** (1) (1980) 1–39.
[67] I.E. Sutherland, SKETCHPAD: A man-machine graphical communication system, MIT Lincoln Labs, Cambridge, MA (1963).
[68] D. Svanaes and E.J. Aas, Test generation through logic programming, *Integr. VLSI J.* **2** (1984) 49–67.
[69] Y. Takamatsu and K. Kinoshita, An efficient test generation method by 10-V algorithm, in: *Proceedings IEEE International Symposium on Circuits and Systems*, Kyoto, Japan (1985) 679–682.
[70] Y. Tohma and K. Goto, Test generation for large-scale combinational circuits by using Prolog, in: *Proceedings 6th International Conference on Logic Programming*, Lisbon, Portugal (1987).
[71] E. Trischler and M. Schulz, Applications of testability analysis to ATG: methods and experimental results, in: *Proceedings IEEE International Symposium on Circuits and Systems*, Kyoto, Japan (1985) 691–694.
[72] P. Van Hentenryck, *Constraint Satisfaction in Logic Programming*, Logic Programming Series (MIT Press, Cambridge, MA, 1989).
[73] P. Van Hentenryck and Y. Deville, The cardinality operator: a new logical connective and its application to constraint logic programming, in: *Proceedings Eighth International Conference on Logic Programming (ICLP-91)*, Paris, France (1991).
[74] P. Van Hentenryck and T. Graf, Standard forms for rational linear arithmetics in constraint logic programming, in: *Proceedings International Symposium on Artificial Intelligence and Mathematics*, Fort Lauderdale, FL (1990).
[75] P Van Hentenryck, V. Saraswat and Y. Deville, Constraint Logic Programming over Finite Domains: the Design, Implementation, and Applications of `cc(FD)`, Tech. Rept., Brown University, Providence, RI (1992).
[76] P. Varma and Y. Tohma, Protean, a knowledge based test generator, in: *Proceedings IEEE 1987 Custom Integrated Circuits Conference*, Portland, OR (1987).
[77] D. Waltz, Generating semantic descriptions from drawings of scenes with shadows, Tech. Rept. AI271, MIT, Cambridge, MA (1972).
[78] T.W. Williams, VLSI Testing, Advances in CAD for VLSI **5** (North-Holland, Amsterdam, Netherlands, 1986).