

Constraint Satisfaction using Constraint Logic Programming

Pascal Van Hentenryck¹

Helmut Simonis²

Mehmet Dincbas³

Technical Report No. CS-91-62

November, 1991

¹Brown University, Computer Science Department, Box 1910, Providence, RI 02912 USA

²Cosytec, Parc Club Orsay-University, 4, rue Jean-Rostand, 91893 Orsay Cedex, France

³Cosytec, Parc Club Orsay-University, 4, rue Jean-Rostand, 91893 Orsay Cedex, France

Constraint Satisfaction using Constraint Logic Programming

Pascal Van Hentenryck

Brown University, Box 1910,
Providence, RI 02912 (USA)
Email: pvh@cs.brown.edu

Helmut Simonis and Mehmet Dincbas

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

Abstract

Constraint Logic Programming 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 a short development time and an efficiency comparable to procedural tools (using 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, inside constraint programming languages, computational paradigms underlying the solving of combinatorial search problems. It is motivated by the hope of reducing significantly the development 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 ($\mathcal{NP}$ -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 [3] to name a few. 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 [43]. Constraint processing itself has also been present in many systems related to constraint solving such as REF-ARF [24], Alice [41], (assumption-based) truth maintenance systems (e.g. [18, 16]), and various scheduling and planning systems (e.g. [25]) to name only a few.

The starting point of our research was slightly different however and based on the recognition 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) non-determinism 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. The main reason comes from the passive use of constraints which only test potential values instead of pruning the search space in an active manner [28]. As a consequence, traditional logic programming languages (e.g. Prolog) often lead to “generate and test” or “standard backtracking” approaches which exhibit a pathological behaviour known as thrashing [43].

The design and implementation of early Constraint Logic Programming (CLP) languages such as CHIP [22], CLP($\mathbb{R}$) [38], Prolog II [10], Prolog III [13], was an attempt to preserve the advantages of logic programming while removing their limitations. The fundamental idea behind these languages, replacing unification by constraint solving as the kernel operation of the language, was elegantly captured in the CLP scheme [37]. 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). For instance, CHIP contains constraint systems over finite domains [73], Booleans [7] and rational numbers [31, 75], Prolog III is endowed with constraint systems over Booleans, rational numbers, and lists while CLP($\mathbb{R}$) solves constraints over real numbers. The CLP scheme was further generalized into the cc framework of concurrent constraint programming [54, 64, 65] to accommodate additional constraint operations (e.g. constraint entailment [44]) and new ways of combining them (e.g. implication (or blocking ask) [54] and cardinality [74]). More precisely, the cc framework accommodates all operations on constraints which can be defined as closure operators. The generalization extends significantly the scope of CLP languages by enabling issues such as concurrency, control, and extendibility to be addressed at the language level.

CLP languages¹ support, in a declarative way, the solving of combinatorial search problems using the global search paradigm. The global search paradigm amounts to dividing recursively a problem into subproblems until the subproblems are simple enough to be solved in a straightforward way. It includes, as special cases, implicit enumeration, branch and bound, and constraint satisfaction. It is best contrasted with the local search paradigm

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

which proceeds by modifying an initial configuration locally until a solution is obtained. Both approaches are orthogonal and complementary. The global search paradigm has been used successfully to solve a large variety of combinatorial search problems with a reasonable efficiency (e.g. scheduling [14], graph coloring [40], Hamiltonian circuits [9], microcode labeling [20]). It provides, at the same time, the basis for exact methods as well as approximated solutions (giving rise to the so-called anytime algorithms [15]).

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 where many algorithms have been proposed for the task. We show that it is possible, using constraint logic programming, to design a simple algorithm whose behaviour is similar in spirit to some of the best algorithms for the task and whose efficiency is competitive (to a small constant factor) with specialized implementations of these algorithms. The second problem, car-sequencing, was motivated by an article in *AI Expert* which presented it as a challenge for AI tools [49]. We propose a solution to this problem which can be concisely described in constraint logic programming and whose efficiency allows for the solving of large instances.

The CLP language used to describe the solutions to the above problems is `cc(FD)`, an instance of the `cc` framework over finite domains which is best seen as a successor to the finite domain part of `CHIP`. Both languages support the use of consistency techniques and local propagation in conjunction with (don't know) non-deterministic choices (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 universal quantification) and the availability of constraint entailment as a primitive operation on constraints. `cc(FD)` generalizes in an elegant way, and removes the need for, several features and constraints of `CHIP` which were difficult to justify theoretically. As a consequence, it provides additional operational expressiveness, flexibility, and efficiency and allows the user to tackle problems such as disjunctions of constraints and the definition of primitive constraints. Preliminary solutions of the two problems described in this paper were first expressed in `CHIP` (see [58] and [19]). The presentation proposed in this paper subsumes them, both in the algorithmic methods which are more advanced than previously 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, the overview of the language is limited to those aspects which are of direct relevance to the two problems considered. Important combinators such as constructive disjunctions and universal quantification are omitted here but can be found in [76]. Sections 4 and 5 present respectively the test generation and car sequencing problems. For each application, we describe in detail how the problem can be stated, how constraint solving is performed and we report a number of experimental results and comparisons. Section 5 contains the conclusion of this research.

```

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

```

Figure 1: An Outline of the Syntax

2 Overview of cc(FD)

In this section, we present an overview of (relevant parts of) $cc(FD)$. The presentation here is informal and should give a preliminary understanding of the system. A more formal presentation, following closely the style of operational semantics used in and promoted by [54], is described in the appendix.

To present informally $cc(FD)$, we proceed 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 respectively constraint entailment, the implication combinator and the cardinality combinator. Section 2.6 discusses the detail of constraint solving in $cc(FD)$. Note that the presentation separates the generic aspects of the language from the detail of its constraint solver. This indicates that the combinators are general-purpose.

2.1 Syntax

An outline of the syntax of $cc(FD)$ program is given in Figure 1.

A $cc(FD)$ is a set of clauses, each clause being made of a head and a body. A head is an atom, i.e. an expression of the form $p(t_1, \dots, t_n)$ where $t_1, \dots, t_n$ 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), either a goal (procedure call), either a constraint (constraint solving), either 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, s , atoms by letters H, B , goals by the letter G , integers by the letters l, u, v , all of them possibly subscripted or superscripted. To illustrate the operational semantics of (part of) $cc(FD)$, we make use of a simple program depicted in Figure 2.

2.2 The CLP Scheme

The operational semantics of the CLP scheme is a simple (at least from a conceptual standpoint) 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

```

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.

```

Figure 2: A simple Program

1. *a goal part*: the conjunction of goals that remains to be solved;
2. *a constraint store*: the set of constraints accumulated up to this point of the execution.

Initially the constraint store is empty and the goal part is the initial goal. In the following, we denote the computation state by pairs $\langle G \sqcap \sigma \rangle$ 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 of computation state is

$\langle q(X,Y,Z) \sqcap X,Y,Z \in \{0,\dots,10\} \ \& \ X \geq Z+3 \ \& \ Y \leq Z \rangle$.

A *computation step* (i.e. the transition from a computation state to another computation state) 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 which can be used to resolve the atom; this clause should 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 which is satisfiable 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

$$\langle q(X,Y,Z) \square X,Y,Z \in \{0,\dots,10\} \& X \geq Z+3 \& Y \leq Z \rangle$$

a computation step can be performed using clause 2 of q to obtain a new computation state

$$\langle Z \geq Y+2 \square X,Y,Z \in \{0,\dots,10\} \& X \geq Z+3 \& Y \leq Z \rangle.$$

Another computation step leads to the configuration

$$\langle \epsilon \square X,Y,Z \in \{0,\dots,10\} \& X \geq Z+3 \& Y \leq Z \& Z \geq Y+2 \rangle$$

since the resulting constraint store is satisfiable.

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 for 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

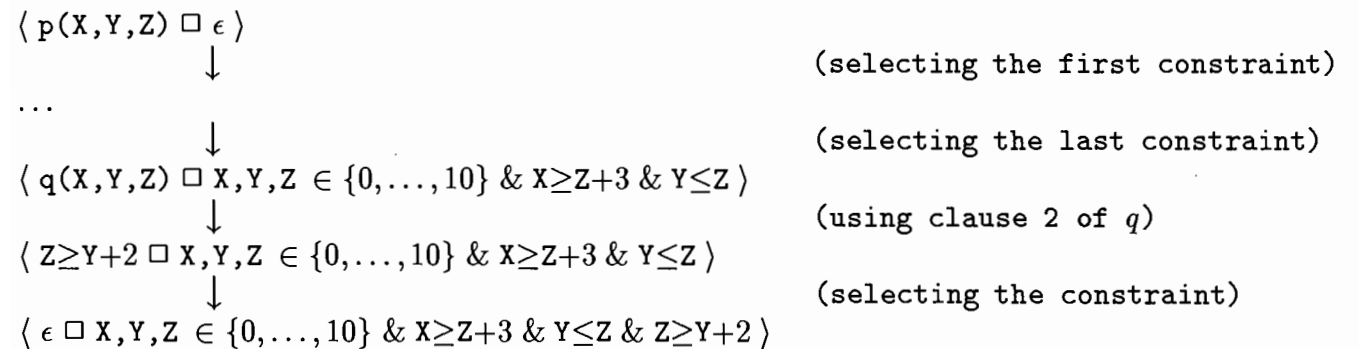
$$\langle \epsilon \square X \in \{5,\dots,10\} \& Y \in \{0,\dots,5\} \& Z \in \{2,\dots,7\} \& X \geq Z+3 \& Y \leq Z \& Z \geq Y+2 \rangle.$$

A computation state is *terminal* if

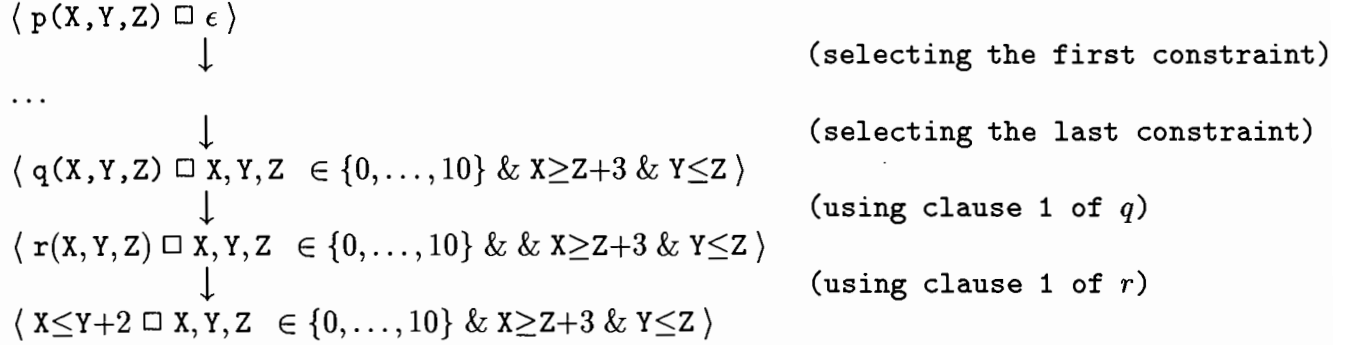
- the goal part is empty;
- no clause can be applied to the selected atom to produce a new computation state or the selected constraint is not satisfiable with the constraint store.

A *computation* is simply a sequence of computation steps ending in a terminal computation state or *diverging*. A finite computation is *successful* if the final computation state has an empty goal. It is *failed* otherwise.

To illustrate computations in a CLP language consider our simple program again. The program has only one successful computation depicted as follows:



The program has also one failed computation:



The last computation state is terminal since the conjunction of constraints

$$X \geq Z+3 \ \& \ Y \leq Z \ \& \ X \leq Y+2$$

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 ordering, 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 will typically use clause 1 for p , then clause 1 for q , and will finally encounter a failure when trying to solve r . Execution will then backtrack to clause 2 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. [56]) by Maher [44] 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.[11, 10, 17, 29, 48]). It is one of the cornerstones of the cc framework where it is used to synchronize concurrently executing agents. It was also used in $CHIP$ (see [22, 32]) inside the `if_then_else` construct and was instrumental in simulating hybrid circuits. Its interest for CLP languages lies in the opportunity to reason about the constraints and to use the gained

²In the following, we will see that the additional combinators of $cc(FD)$ allows for more sophisticated search procedures.

information for achieving pruning. As we will see, it can be used to express non-primitive constraints following general principles from artificial intelligence and operations research.

Constraint entailment can be used in various ways. Both implication and cardinality, the two `cc(FD)` combinators used in our applications, make use of constraint entailment. The implication combinator was introduced by [54] in the context of concurrent logic programming while the cardinality combinator was proposed explicitly for CLP languages in [74].

2.4 The Implication Combinator

Motivation Local propagation is one of the key ideas behind constraint programming languages such as `CONSTRAINTS` [66] and `ThingLab` [3]. Local propagation (or value propagation) amounts to deducing values for some variables given the values 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 (or constraint-driven) computation where goals are suspended (when not enough information is available) and reactivated (when new information require to reconsider them). The purpose of the implication combinator for CLP languages is precisely to achieve this form of behaviour, to generalize it to any constraint system, and to combine it with (don’t know) 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 the operational semantics. The implication $c \rightarrow A$ makes sure 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 flounders, 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 above example originates from the demon declarations of CHIP. The above modelling is best for applications such as diagnostics [61] and test generation [58] of digital circuits.

The first rule expresses that, as soon as the constraint store entails $X = 0$, the constraint $Z = 0$ must be added to the constraint store. Note also the last three rules which actually achieve more than local propagation of values; they also propagate symbolic equalities and one of them is conditional to a symbolic equality. Now the goal $\langle \text{and}(X,Y,Z) \sqcap X = 0 \rangle$ will produce a constraint store $X = 0 \ \& \ Z = 0$, since the goal $\langle X = 0 \rightarrow Z = 0 \sqcap X = 0 \rangle$ reduces to $\langle Z = 0 \sqcap X = 0 \rangle$ and hence to the constraint store $X = 0 \ \& \ Z = 0$. However the goal $\langle \text{and}(X,Y,Z) \sqcap \epsilon \rangle$ 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 which has floundered can be resumed when new information become available in the constraint store. Assume for instance the computation state

$\langle X = 0 \rightarrow Z = 0, T = 0 \rightarrow X = 0 \sqcap T = 0 \rangle$.

The first goal $X = 0 \rightarrow Z = 0$ flounders since $X = 0$ is not entailed by the constraint store. But the second goal can be executed leading eventually to the computation state

$\langle X = 0 \rightarrow Z = 0 \sqcap X = 0 \ \& \ T = 0 \rangle$.

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 assume that we would like to build 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, Y are two input bits, Cin is the carry-in, S is the result bit and C is the carry-out. Now, using the implication combinator to define all logical gates, the query $\text{fa}(X,Y,1,S,0)$ will produce 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 $C1, C2$ 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 zero, which implies that X and Y must be equal because of the first xor-gate. Since X, 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 constraint 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 [45]) but, as should be clear from the presentation, it is not limited to finite domain constraints. 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 $S1$, $S2$ represent the starting dates of the tasks and $D1$, $D2$ 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. task 2 is scheduled after task 1, 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. task 1 is scheduled after task 2, will then be considered. In general, it is better to postpone choices as long as possible. The above constraint can be used in two (symmetric) 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 task 2 cannot be scheduled after task 1 and (2) if the maximal start date of $S1$ is smaller than the minimal start date of $S2$ added to $D2$, then task 1 cannot be scheduled after task 2. The cardinality combinator enables us to express this pruning in a natural way.

Description As mentioned previously, the cardinality combinator has the form

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

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

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

Note that this combinator is quite expressive. A conjunction $c_1 \wedge \dots \wedge c_n$ can be expressed as $\#(n, *, [c_1, \dots, c_n])$ where $*$ is a don't care value. A disjunction $c_1 \vee \dots \vee c_n$ as $\#(1, *, [c_1, \dots, c_n])$ 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( $S_1, D_1, S_2, D_2$ ) :-  
     $\#(1, *, [S_1 + D_1 \leq S_2, S_2 + D_2 \leq S_1])$ .
```

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 ways. Moreover if there is only one way to satisfy it, then the constraints necessary to satisfy it are introduced in the constraint store. To check if there is a way to satisfy the constraint, constraint entailment is used. 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 entails that the first task cannot be scheduled before the second one, then a constraint forcing the second task to be scheduled first is added to the constraint store.

Let us take a simple example to clarify further the behaviour.

$$\begin{array}{c} \langle \#(1, 2, [X=4, Y=10]) \ \& \ X>6 \ \Box \ \epsilon \rangle \\ \downarrow \\ \langle \#(1, 2, [X=4, Y=10]) \ \Box \ X>6 \rangle \\ \downarrow \\ \langle \#(1, 2, [Y=10]) \ \Box \ X>6 \rangle \\ \downarrow \\ X>6 \ \& \ Y=10 \end{array}$$

The above example contains a cardinality combinator requiring that $X = 4$ or $Y = 10$ be true. Initially none of these two constraints are entailed by the constraint store. Also none of their negations is entailed by the constraint store. So the execution of the cardinality combinator flounders. The second goal $X > 6$ is selected which implies that $X \neq 4$ is entailed by the constraint store. There is only one way now to satisfy the cardinality combinator, i.e. adding the constraint $Y=10$ to the constraint store.

2.6 Constraint System

In this section, we describe the constraint part of $cc(FD)$. The presentation is necessarily informal due to the space requirements and the focus of the paper.

Syntax

Definition 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. $t_1 + t_2$, $t_1 * t_2$, $t_1 - t_2$ are arithmetic terms if t_1, t_2 are arithmetic terms.

The primitive constraints of the language are as follows.

Definition 2 A primitive constraint in $cc(FD)$ is of the form

$$x \in \{v_1, \dots, v_n\} \text{ or } t_1 \delta t_2$$

where x is a variable, $v_1, \dots, v_n$ are natural numbers, t_1, t_2 are arithmetic terms and $\delta \in \{>, \geq, =, \neq, \leq, <\}$. Constraints of the first type are called domain constraints while constraints of the second type are called arithmetic constraints.

Note that we require in $cc(FD)$ that each variable appearing in a arithmetic constraint also occurs 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 are required to 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 3 A constraint $c(x_1, \dots, x_n)$ is arc-consistent wrt $D_1, \dots, D_n$ if, for each variable x_i and value $v_i \in D_i$, there exist values $v_1, \dots, v_{i-1}, v_{i+1}, \dots, v_n$ in $D_1, \dots, D_{i-1}, D_{i+1}, \dots, D_n$ such that $c(x_1, \dots, x_n)$ holds.

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

Definition 4 A constraint $c(x_1, \dots, x_n)$ is arc-entailed by $D_1, \dots, D_n$ iff, for all values $v_1, \dots, v_n$ in $D_1, \dots, D_n$, $c(x_1, \dots, x_n)$ 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 before where consistency has been replaced by the weaker notion of arc-consistency and entailment by the weaker notion of arc-entailment. Enforcing arc-consistency does not produce a decision procedure

in general (see [21] for subclasses having that property). In conjunction with nondeterminism, it produces the kind of languages advocated in [43]. Arc-consistency algorithms have been intensively studied in the past [46, 43, 77, 45]. Note however that, given the primitive constraints considered in $cc(FD)$, more efficient algorithms can be exhibited. For instance, when binary constraints are considered, arc-consistency can be enforced in $O(ed)$ where e is the number of constraints and d is the size of the largest domain [21].

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 dividing the primitive constraints into two classes: (1) basic constraints (i.e. those which allow an efficient decision procedure) and (2) non-basic constraints which are defined in terms of the combinators³. The main benefit of investigating the formal semantics has been the identification of a number of new combinators (e.g. constructive disjunction and universal quantification) which support, at the language level, pruning principles that were 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 to evaluate general problem solving techniques, since many special purpose methods have been developed in this area. This gives the possibility of comparing different approaches, using widely available benchmark examples. CLP in general, and CHIP in particular, have been applied to a number of problems from digital circuit design, including formal verification [63], diagnosis [61], synthesis [62, 2] as well as simulation of hybrid circuits [32]. The use of CLP for test generation has been discussed before [57, 58, 60]. The method described here is based on [59]. We show that $cc(FD)$ allows a simple and declarative expression of test generation as a constraint satisfaction problem. Moreover, using the implication operator to define demons, an efficient test generation algorithm is obtained which requires only a fraction of the development effort necessary with conventional approaches.

3.1 Problem Statement

VLSI chips are produced by complex processes. Many possible causes can introduce faults in a circuit and 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 only some percent for a new fabrication process. The manufacturer on the other hand aims at selling chips with a low *defect level*, i.e. a low percentage of faulty chips which pass quality control. *Test generation* is the process of defining which tests should be applied to a circuit in order to detect 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. A consequence of this model

³These constraints do not need to be considered as primitive constraints the language since they can be defined at the language level. They are only provided for convenience.

is the necessity to find a very high percentage of all faults in order to obtain a low defect level for a process with a low yield.

3.1.1 Fault Models

Many different physical failures can occur in a circuit and the only way for testing all possible faults would be an exhaustive test of all circuit behaviours 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 will 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, but *not all*, device faults inside a VLSI circuit. It has been shown that a test set which 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 also use this model and, in the rest of the section, *fault coverage* should be read 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 all 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 problem of ATPG is conceptually split into two subproblems: *test generation* and *fault simulation*. *Test generation* amounts to finding a test which detects a certain fault for some component inside the circuit. *Fault simulation* detects which faults are covered by a 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 [4]:

Setup: To test a fault at the output of a certain gate, it is necessary to ensure a different behaviour between the good and the faulty circuit for this signal. This can be achieved by *controlling the gate*, i.e. by applying certain signals to the input of the circuit.

Propagation: It is not enough to create an internal difference in the behaviour 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.

Justification: The last step assigns values to all signals in the circuit in order to satisfy the conditions enforced by the setup and the propagation step.

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 the basic elements can be implemented as demons using the implication operator. Section 3.2.4 presents the basic test generation program. 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 also 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* which describe components and modules, and the interaction between them. A general description of a full-adder can be expressed 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 would be expressed as follows.

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

In this description, the first argument of a predicate contains the operation mode. For instance, the mode could be 'test' (for test generation), 'simul' (for circuit simulation), 'time' (for delay time computation). It is used to distinguish between several user-defined operation modes. The second argument of a predicate is used to assign a unique identifier for each part (module or basic component) of the circuit. Thus a hierarchical naming convention can be easily implemented. The other arguments represent the inputs and outputs of the components. Note that no distinction is necessary between inputs and outputs. Multiple

internal connections between components are represented with *fanout points*, since they are of special interest in test generation. Although for most other purposes it suffices to represent connections by shared logical variables, it is better to use literals linking distinct variables in test generation.

The full-adder can now be used in other circuit descriptions. Parametrized libraries of modules can be generated using recursive descriptions. This kind of hierarchical description of circuits follows the style of logic programming in a 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 for various applications including simulation, formal verification and fault diagnosis as shown in [60]. Similar ways of describing hardware in logic programming have been described in [68, 33, 23, 34, 12].

3.2.2 ATPG as a Constraint Satisfaction Problem

Our method is based on a description of the test generation problem as a consistent labeling problem. We use six symbolic values, 0, 1, d, dnot, e and enot. d and e values both denote signal differences between the good and the faulty circuit.⁴ dnot and enot are respectively the negations of d and e. The d values are particularly important. d is assigned to the gate under test and d or dnot need to be propagated to a primary output to be able to observe the gate. Hence, to find out if the gate behaves correctly, it suffices to examine the value of the primary output associated with d or dnot. The e values are introduced because of fanout points. At every fanout point, only one fanout stem can have the d-value. The others have an e-value. The main difference between d,dnot and e,enot lies in the way they are propagated by the constraints. The constraints aim at propagating d,dnot towards the primary outputs while no such effort is spent for e,enot. This distinction makes sure that the algorithm be complete (i.e. could find a test if one exists) which will not be true if a five-value logic is used instead.

The definitions of the logical gates prohibit certain input combinations together with d-values in order to enforce propagation. For instance, the second input of an and-gate can only have values 1 or e. All other inputs would block the d-path and are therefore prohibited. The following table shows the admissible combinations for the and, xor and not gates. A minus sign - denotes that an input combination is not allowed.

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	$\bar{e}$

⁴In cc(FD), the symbolic values are associated to natural numbers. We use constants here to simplify presentation.

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

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

The possible values for a fanout point are given by the following predicate definition:

```
% 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).
```

Test generation is then performed by the following method. Variables 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 table above). The gate under test will have a d as output and suitable inputs to control the gate. The important 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* (g.u.t.) to a primary output is guaranteed by the constraints (i.e. by any assignment satisfying the constraints). This is the main difference with the classical ATPG algorithms [53, 30, 27] which use a five valued logic. They rely on a global control strategy to create the d-path and to choose between alternatives. Note also that the problem 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. To obtain an effective approach, we make use mainly of two features of cc(FD): domain constraints and the implication operator. Each line in the circuit is associated to a variable constrained to take one of the six possible values. In addition, the primary inputs are constrained to take 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 the one presented in the description of the implication operator. For instance, for an and-gate, the demon is defined as follows.

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

Note that the implications use both equations, disequations, and domain constraints to reduce the search space by removing values from the variables. Also, each implication solves the constraint, i.e. if an implication has been 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 constraints will flounder until an assignment is made to one input either by propagation or by a labeling routine.

3.2.4 The Basic ATGP Program

We are now in position to present the basic program. For each gate, we define two clauses. The first clause handles the case of the gate under test and control the gate. It assigns a d-value to its output and state a constraint between the inputs such that the output is *set up* correctly. The latter constraint is simply expressed using 0-1 demons as was described in the presentation of the implication operator. The second clause handles the general case of a gate not under test, in which case the program makes use of the `and_demon` to observe the result.

To select which gate to test and which fault value to set up, we define a new mode `test` with two arguments. The first contains the label of the g.u.t, the second the fault to test. This enables us to pass the information to all gates without changing our circuit description. The description of an and-gate for test generation purpose is shown in Figure 3.

The complete program for test generation is shown in Figure 4. It uses a circuit description and the predicate definitions from 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 will be

```

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  $\neq$  Gate,                  % it is not the g.u.t.
    Z  $\in$  {0,1,d,dnot,e,enot},        % domain constraint
    and_demon(X,Y,Z).                % use the six valued demon

```

Figure 3: The and-gate Definition for ATPG

```

% 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).

```

Figure 4: The ATPG Program

assigned to 0/1 values by the labeling routine. The predicate `domain_constraints` makes sure to generate suitable domain constraints for the primary inputs.

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

3.2.5 Heuristics

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

Controlability/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, it is possible to continue the d-path along each of the stems. We have to decide which one to follow. In test generation, several measures have been proposed to estimate the difficulty of finding a path from some point inside the circuit to an output [5]. This value, called *observability* measure, can be precomputed by a preprocessing step. For each fanout point, we obtain an ordering for the fanout-stems, and we 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 *controlability* 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 zero, we use this value; in the other case, we set it to one.

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

Labeling The choice of an appropriate labeling routine is crucial for many constraint satisfaction problems. It turns out that this is not as important for test generation. We use a routine which assigns the variables in the given order, but chooses randomly between 0 and 1 for the first assignment. For most of the example circuits tested below, the labeling is performed 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 much time 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 limits the execution time spent on each case. Both methods are rather simple to add to the program.

3.3 Illustration

To illustrate the behaviour of the program, we make use the full-adder circuit described above. We explain the steps required to generate a test for a *sa1* fault at the output of the xor-gate [2] (See Figure 5).

The query to execute is

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

In order to test for a *sa1* fault, the g.u.t. [2] must be set up for the value 0. This assignment entails, by definition of the xor-gate, the unification of both inputs of gate [2] (see Figure 5 where the unification is shown as a double-arrow), which is the least constraint necessary to make sure that the output be zero. 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),

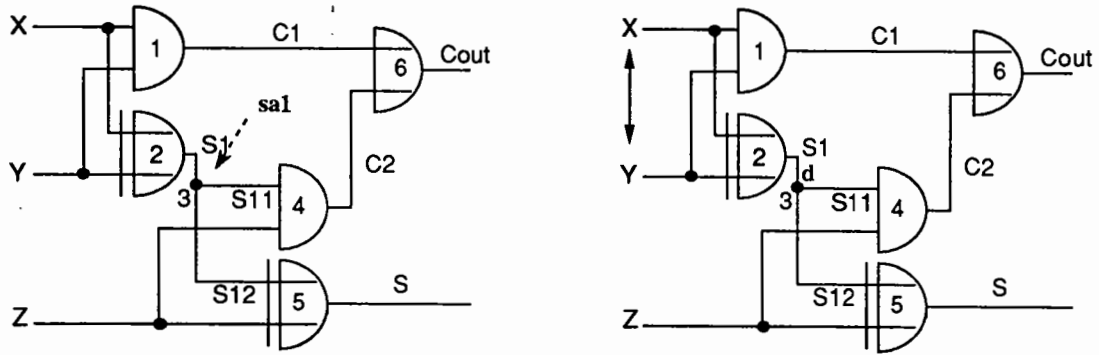


Figure 5: Test Generation Example: Parts (a) and (b)

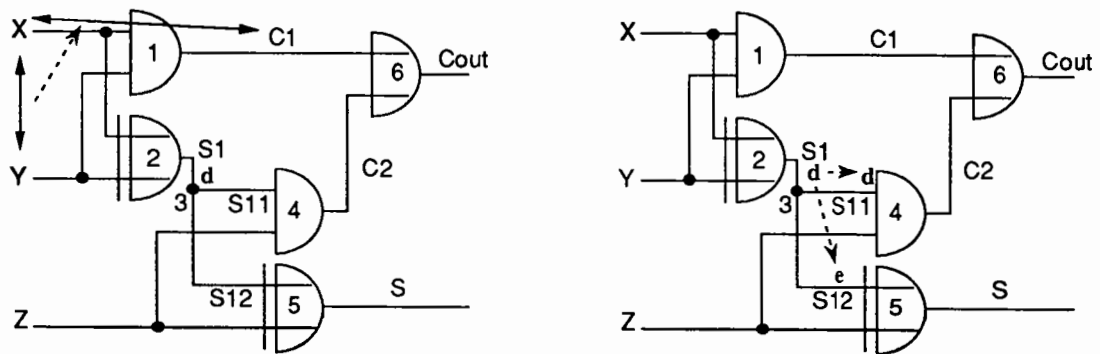


Figure 6: Test Generation Example: Parts (c) and (d)

leading to the unification of X with $C1$ and the removal of d and $dnot$ from $C1$ (see Figure 6). Then the rule for fanout point [3] is executed, creating the constraints $S11 = d$, $S12 = e$ (See Figure 6). This fires another implication for and-gate [4], binding Z to 1 and $C2$ to d , which in turn fires an implication for gate [5], binding S to $enot$ (see Figure 7). The rule

Figure 7: Test Generation Example (e) and (f)

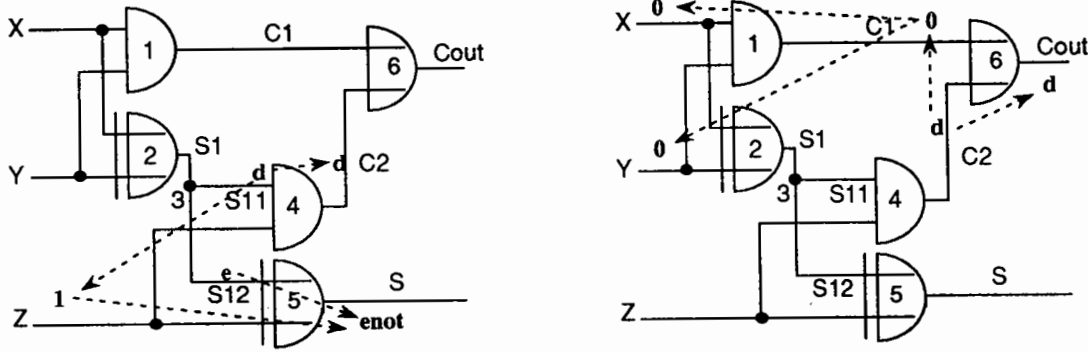
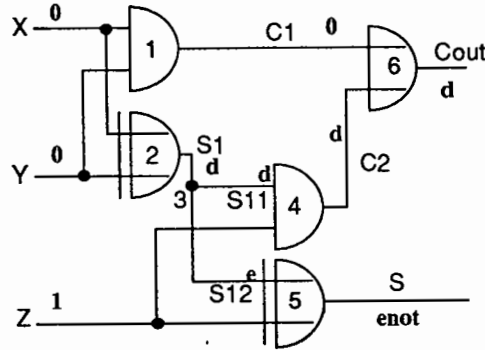


Figure 8: Test Generation Example (g)



for gate [6] now binds C to d and $C1$ to 0, and, by unification, X to 0 and Y to 0 (see Figure 7).

The final solution then is $X = Y = C1 = 0$, $S1 = d$, $S11 = d$, $S12 = e$, $C2 = d$, $C = d$, $Z = 1$, $S = enot$ (as shown in Figure 8). The generated test pattern for the `sa1` fault at the output of gate [2] is $[0,0,1]$.

This example is unusual in that all constraints are solved in the end, i.e. all variables are instantiated to values. For more complex examples, this will not be the case. Some constraints will flounder and wait until variables are instantiated by the labeling procedure in the test predicate.

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

Table 1: Benchmark Results

Name c	Socrates		FAN	D-Alg	AIDSTG		9V		10V(M)	ATWIG	Brglez	FAN	
	%	sec	%	%	%	sec	%	sec	%	%	%	%	sec
432	99.24	5.3	94.7	97.4	99.05	70	99.1	8.1	98.9	95.9	99.24	93.7	3.6
499	98.94	24.9	93.5	68.5	99.29	101	98.9	18.1	98.9	88.0	98.94	99.4	16.2
880	100.00	5.7	100	100	100	107	100	26.3	100	99.2	100	100	1.3
1355	99.49	34.3	93.5	58.2	99.64	301	99.5	72.6	98.7	86.7	97.27	99.5	13.5
1908	99.52	63.1	94.6	95.0	99.59	533	99.6	143.2	99.4	81.9	99.52	99.5	13.5
2670	95.49	61.1	93.2	95.3	96.25	809	95.4	517	93.7	81.1	95.34	95.7	49.4
3540	95.95	89.0	92.0	94.4	95.90	1398	96.1	452	94.7	90.0	95.71	96.0	42.9
5315	98.88	45.4	98.2	98.5	99.21	934	98.9	844	98.6	96.4	98.82	98.9	19.7
6288	99.56	32.8	98.5	99.1	99.48	892	99.6	1039	69.9	99	99.56	99.5	31.7
7552	98.25	243.5	93.7	96.3	98.26	2121	98.1	1446	96.6	92.2	98.19	98.2	118.6

Table 2: Benchmark Comparison

3.4 Computation Results

An evaluation of a test generation method must include experiments with large, realistic circuits. We use the ISCAS benchmark set [36] to test our method. The results show that a constraint based ATPG system, while at the moment not as fast as specialized programs, finds test sets even for large circuits in a reasonable time with a high fault coverage

The benchmark set was defined in 1985 to compare the characteristics of different test generation systems [36]. The results on a SUN 3/260 are given in table 1. The table 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, we show the number of redundant faults detected (Red), the number of aborted fault (Ab), for which the procedure did not find a test or could not detect redundancy, then the fault coverage (%), and the number of test patterns (#) generated. Execution times are shown for test generation only (Time).

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

The average time needed to find one testpattern for each of the example circuits grows nearly linear 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 comparison with 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 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. 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 test pattern even in difficult cases. On the other hand, the current implementation cannot really compete in computational efficiency with the best known conventional programs. But this difference is more a matter of a constant speedup, rather than a difference in complexity.

4 The Car Sequencing Problem

The second application we consider is the so-called car-sequencing problem. It was motivated by an article published in *AI Expert* [49] which submitted it as a challenge for AI technology. We describe a solution using `cc(FD)`.

4.1 Problem Statement

In the car industry, the produced cars may require different options such as air-conditioning, auto-radio, and so on. For production, the cars are placed on an assembly line which moves through the various production units responsible to set up the options. The production units have limited capacity: the option cannot be mounted on each car. More generally speaking, the production units have capacity constraints of the form *r out of s* meaning that, on each sequence of *s* cars, the unit is able to produce at most *r* cars with the option. The car-sequencing problem amounts to finding, given a number of cars with their associated options, a sequencing of the cars satisfying the capacity constraints of the production units.

We illustrate the problem on a simple example. Since cars requiring the same set of options cannot be distinguished here for any useful purpose, we cluster them in classes. Table 9 presents a problem with 5 options, 6 classes, and 10 cars. A “y” in the table means that a particular option is required by the class while a “-” means that the option is not requested. The capacity constraint *r/s* should be read as *r out of s*.

The search space in this problem is made up the possible values for the slots of the assembly line. A “-” in the figures denotes an inconsistent value, a “+” an assigned value while a space denotes a possible value. The assembly line itself is best described by the options selected for each slot. Figures 10 and 11 depict a solution to the simple 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	

Figure 9: A Car Sequencing Example

	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	-	-	+	-	-	-	-	-	-	+

Figure 10: Car Sequencing: 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	-	-	-	+	-	-	-	-	+	-

Figure 11: Car Sequencing: The Assembly Line in a Solution

4.2 Problem Solution

As is typical with finite domains 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 modelling 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 ($n_i \geq 0$) such that the total numbers of cars is $nc = \sum_{i=1}^n n_i$. 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 will assume that *true* is represented 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 ($1 \leq i \leq nc$), we associate a variable S_i denoting the class of cars assigned to the slot. These variables are called the *slot variables* and represents the main output of the program.

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

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 $S_i \in \{1, \dots, n\}$ while 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 are as follows:

$$S_1 \in 1..6, \dots, S_{10} \in 1..6,$$

$$O_1^1 \in 0..1, \dots, O_1^5 \in 0..1, \dots, O_{10}^1 \in 0..1, \dots, O_{10}^5 \in 0..1.$$

4.2.3 Capacity Constraints

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

$$O_i^j + \dots + O_{i+s-1}^j \leq r \quad (1 \leq i \leq nc - s + 1)$$

have to be generated.

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

$$O_1^1 + O_2^1 \leq 1,$$

...

$$O_9^1 + O_{10}^1 \leq 1,$$

while option 2 (2 outof 3) generates the constraints

$$O_1^2 + O_2^2 + O_3^2 \leq 2,$$

$$O_2^2 + O_3^2 + O_4^2 \leq 2,$$

...

$$O_8^2 + O_9^2 + O_{10}^2 \leq 2.$$

A generic 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([0]).
atmost1outof2([01,02|0s]) :-
    01 + 02 ≤ 1,
    atmost1outof2([02|0s]).
```

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

4.2.4 Demand Constraints

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

$$exactly(n_i, [S_1, \dots, S_{nc}], i)$$

has to be generated, where $S_1, \dots, S_{nc}$ 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 they are nc 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 does not produce more

cars from a class than is actually necessary. Hence the above constraints reduce to **atmost** constraints,

$$\text{atmost}(n_i, [S_1, \dots, S_{nc}], i).$$

A constraint **atmost**(N,L,M) holds iff there are atmost 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

$$\text{atmost}(n_i, [S_1, \dots, S_{nc}], i).$$

corresponds to the cardinality formula

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

In other words, the cardinality formula makes sure that atmost n_i constraints in $[S_1 = i, \dots, S_{nc} = i]$ hold, and hence that atmost n_i slots be assigned a car from class i . There are n demand constraints. In the example, the following constraints are generated:

```
#(*, 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,M, makes sure that atmost N elements of the list L be 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 be true.

4.2.5 Link Constraints

Although all constraints seem to have been enforced at this point, there is still an important step missing. The option variables and the slot variables have been left completely unconnected. A slot variable can be assigned a value without influencing its corresponding slot 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 $\text{element}(I, L, V)$ 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

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

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

$$\begin{aligned} &\text{element}(S_1, [1, 0, 0, 1, 1], O_1^1), \\ &\dots \\ &\text{element}(S_1, [0, 0, 1, 0, 0], O_1^5), \\ &\dots \\ &\text{element}(S_{10}, [1, 0, 0, 1, 1], O_{10}^1), \\ &\dots \\ &\text{element}(S_{10}, [0, 0, 1, 0, 0], O_{10}^5). \end{aligned}$$

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

How should a constraint $\text{element}(I, L, V)$ be defined? Obviously, we would like that, as soon as I is given a value, V be 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 can be assigned only the values 1, 4, 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 $\text{element}(I, L, V)$ to be arc-consistent.

To enforce arc-consistency on $\text{element}(I, L, V)$, it is sufficient to generate cardinality constraints of the form

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

where e is a value in L and $i_1, \dots, i_p$ are all the positions in list L whose value is e . In general, a constraint has to 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

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

A simple program can be written that generates 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 [45]).

4.2.6 Basic Program

The basic program can now be presented. It amounts to generating the constraints and to generating values to the slots variables.

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

The arguments of the predicate are the list of slots variables and the data characterizing the instance. The generation of constraints will be responsible for creating as many variables as there are slots in the assembly line, for creating the option variables, and for stating all the abovementioned constraints. Generating the constraints can be done by simple recursive programs (as has been shown in the above presentation) and does not raise any particular difficulty. Assigning a value to the slot variables will produce 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 the Efficiency

The above program provides us with a solution to the car-sequencing problem with a reasonable efficiency. It is however possible to speed up significantly the program by exploiting properties of the solutions and making choices wisely. These two improvements are discussed in the following.

Redundant (Surrogate) Constraints A traditional technique in operations research amounts to generating surrogate constraints, i.e. constraints which are not strictly necessary to guarantee correctness of the application but which performs pruning by exploiting properties that have to be satisfied by the solutions. In other words, the constraints are somewhat redundant semantically but not at all operationally.

In the car-sequencing problem, there is a surrogate constraint which is worth exploiting. Assume that option j has a capacity constraint r out of s . We know that the last s slots can only contain r cars so that the other slots should contain all the remaining cars having that option. If p cars requires option j , we may generate a constraint

$$O_1^j + \dots + O_{nc-s}^j \geq p - r.$$

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

$$O_1^j + \dots + O_{nc-k \times s}^j \geq p - k \times r$$

may be generated.

For instance, in our example, option 1 is requested by 5 cars and has capacity “1 out of 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, the following constraints would be generated.

$$\begin{aligned} O_1^1 + \dots + O_8^1 &\geq 4, \\ O_1^1 + \dots + O_6^1 &\geq 3, \\ O_1^1 + \dots + O_4^1 &\geq 2. \end{aligned}$$

	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	-	-								

Figure 12: Car Sequencing: Search Space After 1 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										

Figure 13: Car Sequencing: The Assembly Line After 1 Choice

The effect of these constraints will be 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 The second improvement we discuss is on the choice process. Following [35], we make use of the first-fail principle in the choice process, that is we try to choose the variable which is most constrained to be instantiated next. In the car-sequencing, this is done by choosing the variable with the smallest domain (i.e. which can be given the smallest number of values) and, in case of equality, the more demanding one in terms of the options.

4.3 Illustration

The above program will first state the constraints and then make choices. After the first choice (i.e. $S_1 = 1$), the search space and the assembly line are depicted in Figures 12 and 13. Now, giving to S_2 its first possible value (i.e. 2) will lead directly to the solution presented at the beginning of the example.

Indeed this choice removes immediately the value 2 for all other slot variables and prevents variables S_3, S_4, S_5 from taking the value 4 because of option 4. This intermediary state is depicted in Figures 14 and 15.

But the surrogate constraints for option 2 requires 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

	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	-	-								

Figure 14: Car Sequencing: The Assembly Line After 2 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		-								

Figure 15: Car Sequencing: Search Space After 2 Choices (Part I)

assignments is to fix all option variables concerning option 2 and to remove possible values from the slot variables. This search space at that stage is depicted in Figures 16 and 17.

At this point, the demand constraints for class 5 comes into play. Two cars of class 5 have to be produced and only two places are left for them. Hence, they are assigned immediately. This leads to the search space depicted at in Figures 18 and 19.

The final step amounts to using the surrogate constraints for option 1 which fix all options concerning option 1 and leads to the solution depicted earlier in this paper. The interesting point here is that a solution was found in two choices without any backtracking.

	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	-	-								

Figure 16: Car Sequencing: The Assembly Line After 2 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		-								

Figure 17: Car Sequencing: The Assembly Line After 2 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	-	-			-			-		

Figure 18: Car Sequencing: Search Space After 2 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		-			-			-		

Figure 19: Car Sequencing: The Assembly Line After 2 Choices (Part III)

n	nc	%-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.

Figure 20: 100 cars sequencing with around 70 % of option utilization

n	nc	%-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.

Figure 21: 100 cars sequencing with around 70 % of option utilization

4.4 Computation Results

A large number of experiments have been run to evaluate the efficiency of the algorithm on various instances of the problem. 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 of 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 n , the particular requirements for the cars, and the utilization of each production unit. In our experiments, n has been varying from 5 to 200, random data has 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 increase further this percentage. For each experiment, large samples of data (around 100) were generated.

When $n < 50$, the program finds a solution in a few seconds and generally with very few backtracking if any. When $n = 50$, the scheduling time is around 15 seconds on a SUN 3/160. Once again, very few backtrackings were needed to reach a solution.

When $n = 100$, the average scheduling time is less than a minute with a utilization percentage around 70 %. Table 20 reports some typical results. The first column is the

n	nc	%-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.

Figure 22: 100 cars sequencing with around 70 % of option utilization

number of cars, the second column indicates the number of different classes, the next five columns indicate the utilization percentage of each option and the last one shows the CPU time required to generate the constraints and find a schedule. Increasing the utilization percentage to 80 % only increases the CPU time by a few seconds. Typical results are shown in Table 21.

When $n = 200$, the average scheduling time is around 5 minutes for an overall utilization percentage of 70%. The average scheduling time does not change when we increase the percentage to 80%. Table 22 reports some typical results.

Note that the potential search space to explore in the last example is 200^{31} . The program has to generate 200 slots variables, 1000 option variables, more than 1000 cardinality constraints and about 3000 numerical constraints. Only the slot variables need to be instantiated to give a solution however. Also an integer programming solution would require more than 7000 variables, of which 6000 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 $n = 400$) although such instance could be constructed since the problem is NP-complete.

5 Conclusion

In this paper, we have shown how to solve two practical combinatorial search problems using `cc(FD)`, a successor of `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 is a problem which 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 an “equivalent” procedural programs. The resulting programs can be easily extended, modified, and/or specialized thanks to the declarative nature of the language. In addition, the resulting algorithms are able to solve large instances of the problems in reasonable time and are competitive with procedural programs to a constant factor. This indicates the potential usefulness of languages such as `cc(FD)`.

One of the key features to solve these problems was the ability to use jointly constraint-solving, constraint entailment, the implication and cardinality combinators as well as the nondeterminism inherent to logic programming.

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

Acknowledgment

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. Interactions with Yves Deville and Vijay Saraswat significantly improved our presentation.

References

- [1] J.A. Abraham. Fault modeling in VLSI. In *VLSI Testing*, volume 5 of *Advances in CAD for VLSI*, chapter 1, pages 1–27. North Holland, Amsterdam, 1986.
- [2] N. Beldiceanu and C. Sechet. Using constraint logic programming for implementing a silicon compiler kernel. Technical Report TR-LP-??, ECRC, Munich, FRG, May 1990.
- [3] A. Borning. The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory. *ACM Transaction on Programming Languages and Systems*, 3(4):353–387, 1981.
- [4] P.S. Bottorff. Test generation and fault simulation. In *VLSI Testing*, volume 5 of *Advances in CAD for VLSI*, chapter 2, pages 29–64. North Holland, Amsterdam, 1986.
- [5] F. Brglez, P. Pownall, and R. Hum. Applications of Testability Analysis: From ATPG to Critical Delay Path Tracing. In *Proc IEEE Int Test Conference 1984*, October 1984.
- [6] F. Brglez, P. Pownall, and R. Hum. Accelerated ATPG and Fault Grading via Testability Analysis. In *Proc IEEE International Symposium on Circuits and Systems*, pages 695–698, Kyoto, Japan, July 1985.
- [7] W. Buttner and H. Simonis. Embedding Boolean Expressions into Logic Programming. *Journal of Symbolic Computation*, 4:191–205, October 1987.
- [8] W.T. Cheng. The Back Algorithm for Sequential Test-Generation. In *Proceedings 1988 IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD88)*, Rye Brook, New York, October 1988. IEEE, Computer Society Press.
- [9] N. Christofides. *Graph Theory: An Algorithmic Approach*. Academic Press, New York, 1975.
- [10] A. Colmerauer, H. Kanoui, and M. Van Caneghem. Prolog, Bases Theoriques et Developpements Actuels. *T.S.I. (Techniques et Sciences Informatiques)*, 2(4):271–311, 1983.
- [11] K.L. Clark and F. Mc Cabe. The Control Facilities of IC-PROLOG. In D. Mitchie, editor, *Expert systems in the micro electronic Age*, pages 122–149. Edinburgh University Press, 1979.
- [12] W.F. Clocksin. Logic Programming and Digital Circuit Analysis. *Journal of Logic Programming*, 4(1):59–82, 1987.
- [13] A. Colmerauer. An Introduction to Prolog III. *CACM*, 28(4):412–418, 1990.

- [14] J. Carlier and E. Pinson. Une Methode Arborescente pour Optimiser la Duree d'un JOB-SHOP. Technical Report ISSN 0294-2755, I.M.A, 1986.
- [15] T. Dean and M. Boddy. An Analysis of Time-dependent Planning. In *Proceedings of the Seventh National Conference On Artificial Intelligence*, pages 49–54, Minneapolis, Minnesota, August 1988.
- [16] J. de Kleer. An Assumption-based TMS. *Artificial Intelligence*, 28:127–162, 1986.
- [17] M. Dincbas and J-P. Lepape. Metacontrol of Logic Programs in METALOG. In *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS'84)*, pages 361–370, Tokyo, Japan, November 1984. ICOT.
- [18] J. Doyle. A Truth Maintenance System. *Artificial Intelligence*, 12:231–272, 1979.
- [19] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving the Car Sequencing Problem in Constraint Logic Programming. In *European Conference on Artificial Intelligence (ECAI-88)*, Munich, W. Germany, August 1988.
- [20] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving Large Combinatorial Problems in Logic Programming. *Journal of Logic Programming*, 8(1-2):75–93, 1990.
- [21] Y. Deville and P. Van Hentenryck. An Efficient Arc Consistency Algorithm for a Class of CSP Problems. In *International Joint Conference on Artificial Intelligence*, Sidney, Australia, August 1991.
- [22] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The Constraint Logic Programming Language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, December 1988.
- [23] K. Eshghi. Application of Meta-Level Programming to Fault Finding in Logic Circuits. In *Logic Programming and its Applications*, pages 208–219. Ablex Publishing Corporation, 1985.
- [24] R.E. Fikes. *A Heuristic Program for Solving Problems Stated as Non-deterministic Procedures*. PhD thesis, Comput. Sci. Dept., Carnegie-Mellon Univ. Pittsburgh, PA, 1968.
- [25] M.S. Fox. Constraint-Directed Search: A Case Study of Job-Shop Scheduling. Technical Report CMU-CS-83-161, Carnegie-Mellon University, December 1983.
- [26] H. Fujiwara. FAN: A Fanout Oriented Test Pattern Generation Algorithm. In *Proc IEEE International Symposium on Circuits and Systems*, pages 671–674, Kyoto, Japan, July 1985.
- [27] H. Fujiwara and T. Shimono. On the Acceleration of Test Generation Algorithms. *IEEE Trans Comp*, C-32(Dec):1137–1144, 1983.
- [28] H. Gallaire. Logic Programming: Further Developments. In *IEEE Symposium on Logic Programming*, pages 88–99, Boston, July 1985. Invited paper.
- [29] H. Gallaire and C. Lasserre. *Metalevel Control for Logic Programs*, volume Logic Programming, pages 173–185. Academic Press, 1982.

- [30] P. Goel. An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits. *IEEE Trans Comp*, C-30(Mar):215–222, 1981.
- [31] T. Graf. Extending Constraint Handling in Logic Programming to Rational Arithmetic. Internal Report, ECRC, Munich, September 1987.
- [32] T. Graf, P. Van Hentenryck, C. Pradelles, and L. Zimmer. Simulation of Hybrid Circuits in Constraint Logic Programming. *Computers and Mathematics with Applications*, 20(9/10):45–56, 1990. Preliminary Version Appeared In *International Joint Conference on Artificial Intelligence*, Detroit, Michigan, August 1989.
- [33] E. Gullichsen. Heuristic circuit simulation using PROLOG. *INTEGRATION, the VLSI Journal*, 3:283–318, 1985.
- [34] R. Gupta. Test-pattern Generation for VLSI circuits in a Prolog Environment. In *Proceedings of the Third International Conference on Logic Programming*, pages 528–535, London, July 1986.
- [35] R.M. Haralick and G.L. Elliot. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence*, 14:263–313, 1980.
- [36] ISCAS. Special Session on ATPG. In *Proc IEEE International Symposium on Circuits and Systems*, pages 663–698, Kyoto, Japan, July 1985.
- [37] J. Jaffar and J-L. Lassez. Constraint Logic Programming. In *POPL-87*, Munich, FRG, January 1987.
- [38] J. Jaffar and S. Michaylov. Methodology and Implementation of a CLP System. In *Fourth International Conference on Logic Programming*, Melbourne, Australia, May 1987.
- [39] M. Kawai, K. Oozeki, M. Takahashi, M. Ono, Y. Ishizaka, and T. Masui. Automatic Test Pattern Generator for Large Combinational Circuits. In *Proc IEEE International Symposium on Circuits and Systems*, pages 663–666, Kyoto, Japan, July 1985.
- [40] M. Kubale and D. Jackowski. A Generalized Implicit Enumeration Algorithm for Graph Coloring. *CACM*, 28(4):412–418, 1985.
- [41] J-L. Lauriere. A Language and a Program for Stating and Solving Combinatorial Problems. *Artificial Intelligence*, 10(1):29–127, 1978.
- [42] A. Liroy. Adaptive Backtrace and Dynamic Partitioning Enhance ATPG. In *Proceedings 1988 IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD88)*, Rye Brook, New York, October 1988. IEEE, Computer Society Press.
- [43] A.K. Mackworth. Consistency in Networks of Relations. *AI Journal*, 8(1):99–118, 1977.
- [44] M.J. Maher. Logic Semantics for a Class of Committed-Choice Programs. In *Fourth International Conference on Logic Programming*, pages 858–876, Melbourne, Australia, May 1987.
- [45] R. Mohr and T.C. Henderson. Arc and Path Consistency Revisited. *Artificial Intelligence*, 28:225–233, 1986.

- [46] U. Montanari. Networks of Constraints : Fundamental Properties and Applications to Picture Processing. *Information Science*, 7(2):95–132, 1974.
- [47] M. Muarkami and H. Kikuchiara. Test Generation for LSI Circuits Using Extended Nine-Valued Method. In *Proc IEEE International Symposium on Circuits and Systems*, pages 675–678, Kyoto, Japan, July 1985.
- [48] L. Naish. *Negation and Control in Prolog*. PhD thesis, University of Melbourne, Australia, 1985.
- [49] B.D. Parrello, W.C. Kabat, and L. Wos. Job-Shop Scheduling Using Automated Reasoning: A Case Study of the Car-Sequencing Problem. *Journal of Automated Reasoning*, 2(1):1–42, 1986.
- [50] G.D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, CS Department, University of Aarhus, 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 Combinational Circuits. In *Proc IEEE International Symposium on Circuits and Systems*, pages 667–670, Kyoto, Japan, July 1985.
- [53] J. Roth. Diagnosis of Automata Failure: A Calculus and a Method. *IBM Journal of Research and Development*, 10:278–291, 1966.
- [54] V.A. Saraswat. *Concurrent Constraint Programming Languages*. PhD thesis, Carnegie-Mellon University, 1989.
- [55] M. Schulz, E. Trischler, and T. Sarfert. Socrates: A Highly Efficient Automatic Test Pattern Generation System. In *Proc Int Test Conference*, Washington D.C., Sept 1987.
- [56] E. Shapiro. The Family of Concurrent Logic Programming Languages. *Computing Surveys*, 21(3):413–510, 1990.
- [57] H. Simonis. Test Generation Using Logic Programming. Technical Report TR-LP-34, E.C.R.C (European Computer-Industry Research Centre), April 1988.
- [58] H. Simonis. Test Generation Using the Constraint Logic Programming Language CHIP. In *Proceedings of the 6th International Conference on Logic Programming*, Lisbon, Portugal, June 1989.
- [59] H. Simonis. ATPG Revisited. Technical Report TR-LP-56, E.C.R.C. (European Computer-Industry Research Centre), August 1990.
- [60] H. Simonis and M. Dincbas. Using an Extended Prolog for Digital Circuit Design. In *IEEE International Workshop on AI Applications to CAD Systems for Electronics*, pages 165–188, Munich, W.Germany, October 1987.

- [61] H. Simonis and M. Dincbas. Using Logic Programming for Fault Diagnosis in Digital Circuits. In *German Workshop on Artificial Intelligence (GWAI-87)*, pages 139–148, Geseke, W.Germany, September 1987.
- [62] H. Simonis and T. Graf. Technology Mapping in CHIP. Technical Report TR-LP-44, E.C.R.C. (European Computer-Industry Research Centre), July 1990.
- [63] H. Simonis, H.N. Nguyen, and M. Dincbas. Verification of Digital Circuits Using CHIP. In G.J. Milne, editor, *Proceedings of the IFIP WG 10.2 International Working Conference on the Fusion of Hardware Design and Verification*, Glasgow, Scotland, July 1988. IFIP, North-Holland.
- [64] V.A. Saraswat and M. Rinard. Concurrent Constraint Programming. In *Proceedings of Seventeenth ACM Symposium on Principles of Programming Languages*, San Francisco, CA, January 1990.
- [65] V.A. Saraswat, M. Rinard, and P. Panangaden. Semantic Foundations of Concurrent Constraint Programming. In *Proceedings of Ninth ACM Symposium on Principles of Programming Languages*, Orlando, FL, January 1991.
- [66] G.J. Sussman and G.L. Steele. CONSTRAINTS-A Language for Expressing Almost-Hierarchical Descriptions. *AI Journal*, 14(1), 1980.
- [67] I.E. Sutherland. SKETCHPAD: a Man-Machine Graphical Communication System. Cambridge, ma, MIT Lincoln Labs, 1963.
- [68] D. Svanaes and E.J. Aas. Test Generation Through Logic Programming. *INTEGRATION, the VLSI Journal*, 2:49–67, 1984.
- [69] Y. Takamatsu and K. Kinoshita. An Efficient Test Generation Method by 10-V Algorithm. In *Proc IEEE International Symposium on Circuits and Systems*, pages 679–682, Kyoto, Japan, July 1985.
- [70] Y. Tohma and K. Goto. Test Generation for Large-Scale Combinational Circuits by using Prolog. In *Proc 6th Conf Logic Programming*, Tokyo, Japan, June 1987.
- [71] E. Trischler and M. Schulz. Applications of Testability Analysis to ATG: Methods and Experimental Results. In *Proc IEEE International Symposium on Circuits and Systems*, pages 691–694, Kyoto, Japan, July 1985.
- [72] P. Varma and Y. Tohma. Protean, A Knowledge Based Test Generator. In *Proc IEEE 1987 Custom Integrated Circuits Conference*, Portland Or, May 1987.
- [73] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. Logic Programming Series, The MIT Press, Cambridge, MA, 1989.
- [74] P. Van Hentenryck and Y. Deville. The Cardinality Operator: A new Logical Connective and its Application to Constraint Logic Programming. In *Eighth International Conference on Logic Programming (ICLP-91)*, Paris (France), June 1991.

- [75] P. Van Hentenryck and T. Graf. Standard Forms for Rational Linear Arithmetics in Constraint Logic Programming. In *International Symposium on Artificial Intelligence and Mathematics*, Fort Lauderdale, Florida, January 1990. To Appear in *Annals of Artificial Intelligence and Mathematics*.
- [76] P. Van Hentenryck, V. Saraswat, and Y. Deville. Constraint Processing in *cc(FD)*. Technical Report (Forthcoming), Brown University, November 1991.
- [77] D. Waltz. Generating Semantic Descriptions from Drawings of Scenes with Shadows. Technical Report AI271, MIT, MA, November 1972.
- [78] T.W. Williams. *VLSI Testing, Advances in CAD for VLSI Vol. 5*. North-Holland, Amsterdam, Netherlands, 1986.

A Appendix: Formalization of the Semantics of cc(FD)

In this section, we formalize, following [54], the operational semantics using a structural operational semantics [50]. The reader interested in a broader and more rigorous handling the semantics can refer to [37] and [64, 65]. [37] contains a complete description of the CLP scheme while [64, 65] describe respectively 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 5 A transition system is a triple $\langle \Gamma, T, \mapsto \rangle$ where Γ is a set of configurations, $T \subseteq \Gamma$ is the set of terminal configurations and $\mapsto \subseteq \Gamma \times \Gamma$ is the transition relation satisfying

$$\forall \gamma \in T, \forall \gamma' \in \Gamma, \gamma \not\mapsto \gamma'$$

The configurations of the transition system are the computation states $\langle G \sqcap \sigma \rangle$. When the goal part is empty, we take the convention of representing the configuration by the constraint part only. Similarly, when the constraint part is empty, we take the convention of representing the configuration by the goal part only. Terminal configurations are simply successful computation states (i.e. constraint stores) or the terminal **flounder** to denote floundering.

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

$$\frac{\begin{array}{l} < \text{condition 1} > \\ \dots \\ < \text{condition n} > \end{array}}{\gamma \mapsto \gamma'}$$

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.

$$\frac{\begin{array}{l} p(s_1, \dots, s_n) : - B_1, \dots, B_m \in P \\ C \models (\exists) (\sigma \wedge t_1 = s_1 \wedge \dots \wedge t_n = s_n) \end{array}}{\langle p(t_1, \dots, t_n) \sqcap \sigma \rangle \mapsto \langle B_1, \dots, B_m \sqcap \sigma \wedge t_1 = s_1 \ \& \ \dots \ \& \ t_n = s_n \rangle}$$

In the above transition rule, $p(t_1, \dots, t_n)$ is the selected atom, P denotes the program, $(\exists)(\psi)$ represents the existential closure of ψ , and the program clause has been renamed

properly not to share any variable with the goal.⁵ The rule expresses formally the first kind of computation steps which was presented informally 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.

$$\frac{\mathcal{C} \models (\exists)(\sigma \wedge c)}{\langle c \sqcap \sigma \rangle \mapsto \sigma \& c}$$

This rule captures what was informally described by the second type of computation steps.

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.

$$\frac{\langle G_1 \sqcap \sigma \rangle \mapsto \langle G'_1 \sqcap \sigma' \rangle}{\begin{array}{l} \langle G_1, G_2 \sqcap \sigma \rangle \mapsto \langle G'_1, G_2 \sqcap \sigma' \rangle \\ \langle G_2, G_1 \sqcap \sigma \rangle \mapsto \langle G_2, G'_1 \sqcap \sigma' \rangle \end{array}}$$

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

These are the only rules which are 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 flounders.

$$\frac{\mathcal{C} \models (\forall)(\sigma \rightarrow c)}{\langle c \rightarrow A \sqcap \sigma \rangle \mapsto \langle A, \sigma \rangle}$$

⁵In recent work [65], Saraswat gives an operational semantics precluding the need for renaming.

$$\frac{\mathcal{C} \models (\forall) (\sigma \rightarrow \neg c)}{\langle c \rightarrow A \Box \sigma \rangle \mapsto \sigma}$$

$$\frac{\begin{array}{l} \mathcal{C} \models \neg(\forall) (\sigma \rightarrow c) \\ \mathcal{C} \models \neg(\forall) (\sigma \rightarrow \neg c) \end{array}}{\langle c \rightarrow A \Box \sigma \rangle \mapsto \text{flounder}}$$

A.3 The Cardinality Combinator

The precise behaviour of the combinator can be described by the following transition rules taken from [74].

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

$$\frac{l \leq 0 \wedge n \leq u}{\langle \#(l, u, [c_1, \dots, c_n]) \Box \sigma \rangle \mapsto \sigma}$$

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

$$\frac{\begin{array}{l} l \leq u \wedge l = n \\ \mathcal{C} \models (\exists) (\sigma \wedge c_1 \wedge \dots \wedge c_n) \end{array}}{\langle \#(l, u, [c_1, \dots, c_n]) \Box \sigma \rangle \mapsto \sigma \& c_1 \& \dots \& c_n}$$

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

$$\frac{\begin{array}{l} l \leq u \wedge u = 0 \\ \mathcal{C} \models (\exists) (\sigma \wedge \neg c_1 \wedge \dots \wedge \neg c_n) \end{array}}{\langle \#(l, u, [c_1, \dots, c_n]) \Box \sigma \rangle \mapsto \sigma \& \neg c_1 \& \dots \& \neg c_n}$$

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.

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

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:

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

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 flounder as none of the constraints can be decided upon (for entailment) wrt the constraint store.

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

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

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

$$\begin{array}{l} \langle \#(1, 2, [X=4, Y=10]) \& X>6 \square \epsilon \rangle \\ \quad \downarrow \text{conjunction} \\ \langle \#(1, 2, [X=4, Y=10]) \square X>6 \rangle \\ \quad \downarrow \text{negative reduction} \\ \langle \#(1, 2, [Y=10]) \square X>6 \rangle \\ \quad \downarrow \text{positive satisfaction} \\ X>6 \& Y=10 \end{array}$$

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 $\mapsto^*$ the transitive closure of $\mapsto$ and say that a configuration γ diverges in program P if there exists an infinite sequence of transitions $P \vdash \gamma \mapsto \gamma_1 \mapsto \dots \mapsto \gamma_i \mapsto \dots$

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

$$\begin{aligned} SS[P] &= \{G \mid P \vdash G \xrightarrow{*} \sigma\} \\ DS[P] &= \{G \mid G \text{ diverges in } P\} \\ FLS[P] &= \{G \mid P \vdash G \xrightarrow{*} \textit{flounder}\} \end{aligned}$$

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

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

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

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

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

Contents

1	Introduction	1
2	Overview of $cc(FD)$	4
2.1	Syntax	4
2.2	The CLP Scheme	4
2.3	Constraint Entailment	7
2.4	The Implication Combinator	8
2.5	The Cardinality Combinator	10
2.6	Constraint System	11
3	Test Pattern Generation	13
3.1	Problem Statement	13
3.1.1	Fault Models	14
3.1.2	Test Generation and Fault Simulation	14
3.2	Problem Solution	15
3.2.1	Circuit Description	15
3.2.2	ATPG as a Constraint Satisfaction Problem	16
3.2.3	Gates as Demons	17
3.2.4	The Basic ATPG Program	18
3.2.5	Heuristics	19
3.3	Illustration	20
3.4	Computation Results	23
4	The Car Sequencing Problem	24
4.1	Problem Statement	24
4.2	Problem Solution	26
4.2.1	Problem Variables	26
4.2.2	Domain Constraints	26
4.2.3	Capacity Constraints	27
4.2.4	Demand Constraints	27
4.2.5	Link Constraints	28
4.2.6	Basic Program	29
4.2.7	Improving the Efficiency	30
4.3	Illustration	31
4.4	Computation Results	34
5	Conclusion	35
A	Appendix: Formalization of the Semantics of $cc(FD)$	42
A.1	The CLP Scheme	42
A.2	The Implication Combinator	43

A.3 The Cardinality Combinator	44
A.4 Operational Semantics	45