

A Declarative Encoding of Telecommunications Feature Subscription in SAT

Michael Codish

Department of Computer Science
Ben-Gurion University of the Negev
Beer-Sheva, Israel
mcodish@cs.bgu.ac.il

Samir Genaim

DSIC, Facultad de Informática
Complutense University of Madrid
Madrid, Spain
samir.genaim@fdi.ucm.es

Peter J. Stuckey

National ICT Australia
Department of Comp Sci and Soft Eng
University of Melbourne, Australia
pjs@cs.mu.oz.au

Abstract

This paper describes the encoding of a telecommunications feature subscription configuration problem to propositional logic and its solution using a state-of-the-art Boolean satisfaction solver. The transformation of a problem instance to a corresponding propositional formula in conjunctive normal form is obtained in a declarative style. An experimental evaluation indicates that our encoding is considerably faster than previous approaches based on the use of Boolean satisfaction solvers. The key to obtaining such a fast solver is the careful design of the Boolean representation and of the basic operations in the encoding. The choice of a declarative programming style makes the use of complex circuit designs relatively easy to incorporate into the encoder and to fine tune the application.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Constraint and logic languages; D.1.6 [*Software*]: Programming Techniques—Logic Programming; I.2 [*Computing Methodologies*]: Artificial Intelligence

Keywords SAT solving, telecommunications feature subscription, declarative modelling

General Terms Algorithms, Experimentation

1. Introduction

Modern telecommunications enable a customer to subscribe to services selected from a catalog of features. Familiar features include: speed dialing, call waiting, three-way calling, caller identification, call screening, call announce, call blocking, call forwarding, follow-me forwarding, and may more. Large PSTN (Public Switched Telephone Network) switches are reported, for example in (Bond et al. 2004), to involve hundreds or even thousands of features. A configuration is a sequence of selected features. The configuration of a feature subscription is often personalized based on preferences provided by the customer and constraints imposed by the provider to prevent undesirable feature interactions at runtime. For example, the provider may have constraints making a call-logging feature incompatible with call-forwarding-unconditional, and that a do-not-disturb feature may not be selected without first

sequencing the call-logging feature as otherwise calls may be completely lost. A customer may prefer time-dependent-routing over terminating-call-screening if both are available.

Typically, both customer preferences as well as the constraints imposed by the provider are expressed as precedence or exclusion constraints on features. A precedence constraint, $f > g$, means that feature f is only possible if feature g is sequenced before f . An exclusion constraint, $f \diamond g$, means that f and g cannot both be subscribed to and may be formulated using precedence constraints as $f > g \wedge g > f$. When the subscription requested by a user is inconsistent, due to cycles in the precedence constraints, one problem is to find an optimal relaxation which is consistent.

This problem is a variant of the maximum acyclic subgraph problem (MAS): given a directed graph (V, E) and a parameter m , is there a subset $E' \subseteq E$ with $|E'| \geq m$ such that E' is acyclic. An alternative (dual) formulation of the maximum acyclic subgraph problem is the feedback arc set problem (FAS) which is on the list of 21 problems presented by Karp (Karp 1972) in 1972 exhibiting the first NP complete problems. Applications involving MAS are abundant and many research papers address the topic of approximation algorithms and identifying special cases for which there exist efficient algorithms. This paper addresses the general situation in which an exact solution is required and graphs do not belong to a class for which an efficient algorithm is known.

Propositional satisfiability (SAT) solvers are becoming remarkably powerful and there is an increasing number of papers which propose encoding hard combinatorial (finite domain) problems in SAT. The increasing power and success of modern SAT solvers is due to a combination of several techniques (see e.g. (Gomes et al. 2008)). Two main factors are the highly optimized propagation mechanisms based on watched literals (Moskewicz et al. 2001) and the incorporation of efficient conflict-clause learning algorithms based on the UIP scheme (Zhang et al. 2001).

We mention propagation techniques because when choosing an encoding we will prefer a representation that better supports propagation. We mention conflict-clause learning because when iterating with a SAT solver to optimize the selected solution we should be careful to reuse learned clauses from previous iterations.

In recent research, described in (Lesaint et al. 2008), the authors formalize the telecommunications feature subscription configuration problem and prove that the complexity of finding an optimal relaxation is NP-hard. That paper compares three techniques to address the problem using: constraint programming, encoding to Boolean satisfaction (SAT), and integer linear programming. The authors present a series of experiments and conclude that the constraint programming approach is able to scale well compared to the other approaches.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PPDP'09, September 7–9, 2009, Coimbra, Portugal.

Copyright © 2009 ACM 978-1-60558-568-0/09/09...\$5.00

This paper reexamines the encoding of telecommunications feature subscription configuration problems to SAT. Our approach to the encoding leads to a scalable solution which is considerably faster than the three techniques reported in (Lesaint et al. 2008). Given the directed graph (V, E) and parameter m , our encoding involves three main components to construct a propositional formula, satisfiability of which identifies $E' \subseteq E$ with $|E'| \geq m$ such that E' is acyclic:

1. To express acyclicity, each selected edge $(u, v) \in E'$ imposes a constraint of the form $u > v$ where u and v are viewed as integer values and $>$ is interpreted as the standard order on integers.
2. To express that $|E'| \geq m$, each edge $e \in E$ is interpreted as a Boolean variable which indicates if e is selected to be in E' or not. The set of these variables in E are summed to an integer value m' , indicating the size of E' , which must satisfy $m' \geq m$.
3. To find a maximum value of m for which an instance has a solution we need to solve a maximization problem. The SAT solving technology we use for solving propositional formulae only answers satisfaction questions. Hence we need to encode the maximization problem as a series of satisfaction problems. There are a number of ways of doing this.

A key design decision is how to represent integer values and how to encode the elementary constraints such as total/partial orders, summing of weighted bits, addition of integer values and comparison of integer values. Taking a high-level declarative approach facilitates the encoding and combination of a wide variety of low-level options for these design choices. We illustrate the encodings for the main components and demonstrate how complex circuit designs take a clear form when expressed declaratively. An experimental evaluation indicates that our approach leads to an encoding which is much faster than previous approaches based on the use of SAT solvers and which is highly competitive with published results that use constraint programming and integer linear programming.

A preliminary version of this paper (Codish et al. 2008a) was presented as a Short Paper at ICLP 2008. This paper provides the full details. The contributions of this paper are:

- A full description of the encodings as logic programs;
- A new and effective encoding of acyclicity using unary number representation;
- A careful comparison of the effects of different encoding choices on the quality of the encoding; and
- A final encoding that improves on the results of the preliminary version (Codish et al. 2008a) by 1-2 orders of magnitude, and is considerably faster than the previous best published results (Lesaint et al. 2008).

This paper is not about theorems and proofs. It is about how to put together a collection of well studied techniques from the area of circuit design and apply them in the context of SAT encoding. While the existence of an encoding to SAT follows from Cook's theorem, small improvements in the encoding can make large differences when SAT solving. The declarative encoding of circuits to SAT gives us the opportunity to explore which circuit designs are best for Boolean satisfiability, as opposed to, for use in hardware. The trade-offs are quite different. In SAT solving the circuits are omni-directional, and we are interested in the power of unit propagation, often principally in the reverse direction (from outputs to inputs). Each new internal wire is a new propositional variable which can increase the potential search space significantly. Considerations such as gate layout, wiring complexity and circuit fan-out

are irrelevant to SAT encoding, whereas in hardware design they are important considerations.

The remainder of the paper is organized as follows. In the next section we give a full statement of the telecommunications feature subscription problem. In Section 3 we first briefly introduce our notation for Boolean problems, and then discuss how we represent and construct Boolean satisfiability problems in a declarative manner using Prolog. We illustrate this using adder circuits. In Section 4 we give the basic encoding of the feature subscription problem into SAT, leaving unspecified the crucial details of how to encode acyclicity, sum weighted bits, and manage optimization, which are explored later. In Section 5 we explain various options for encoding the acyclicity constraint in SAT. In Section 6 we discuss various options for encoding the summing of bits in SAT, before extending this to weighted bits. In Section 7 we discuss methods for managing optimization with a SAT solver. In Section 8 we give experimental results showing the effect of various choices in the mapping to SAT, and compare against the previous solutions to this problem. Finally in Section 9 we conclude.

2. Problem Statement

This section presents the formal statement of the telecommunications feature subscription configuration problem and is taken (with slight modification) from (Lesaint et al. 2008).

Let F denote a finite set of features. For $f_i, f_j \in F$ a *precedence constraint* $(f_i > f_j)$ indicates that f_i can appear only after f_j . An *exclusion constraint* $(f_i \diamond f_j)$ between f_i and f_j indicates that f_i and f_j cannot appear together in a sequence of features, and is equivalent to the conjunction of $(f_i > f_j)$ and $(f_j > f_i)$. A *catalog* is a pair $\langle F, P \rangle$ with F a set of features and P a set of precedence constraints on F . A *feature subscription* S of a catalog $\langle F_c, P_c \rangle$ is a tuple $\langle F, C, U, W_F, W_U \rangle$ where $F \subseteq F_c$ is the set of features selected from F_c , C is the projection of P_c on F , i.e., $C = \{(f_i > f_j) \in P_c \mid f_i, f_j \in F\}$, U is a set of user defined precedence constraints on F , and $W_F: F \rightarrow \mathcal{N}$ and $W_U: U \rightarrow \mathcal{N}$ are maps which assign weights to features and user precedence constraints. The value of S is defined by

$$Value(S) = \sum_{f \in F} W_F(f) + \sum_{p \in U} W_U(p)$$

The weight associated with a feature or a precedence constraint signifies its importance for the user.

A feature subscription $\langle F, C, U, W_F, W_U \rangle$ of some catalog $\langle F_c, P_c \rangle$ is *consistent* if and only if the directed graph $\langle F, C \cup U \rangle$ is acyclic. Checking for consistency of a subscription is straightforward using topological sort as described in (Lesaint et al. 2008). If a feature subscription is inconsistent then the task is to relax it and to generate a consistent one with maximum value. A *relaxation* of a feature subscription $S = \langle F, C, U, W_F, W_U \rangle$ is a consistent subscription $S' = \langle F', C', U', W_{F'}, W_{U'} \rangle$ such that $F' \subseteq F$, C' is the projection of C on F' , U' is a subset of the projection of U on F' , $W_{F'}$ is the restriction of W_F to F' , and $W_{U'}$ is the restriction of W_U to U' . We say that S' is an *optimal relaxation* of S if there does not exist another relaxation S'' of S such that $Value(S'') > Value(S')$. In (Lesaint et al. 2008), the authors prove that finding an optimal relaxation of a feature subscription is NP-hard. This is the problem addressed in this paper. We take a declarative approach to SAT encoding. An encoding is a Prolog program which maps a problem instance to a corresponding formula in conjunctive normal form. This formula is then used to find a solution to the instance by application of a SAT solver.

3. A Declarative Approach to Encoding

Most SAT solvers assume as input a propositional formula in conjunctive normal form (CNF). That is a conjunction of disjunctions

```

adder([], [], C, [C], Cnf-Cnf) .
adder([], [X|Xs], Cin, [Z|Zs], Cnf1-Cnf3) :-
    half_adder(X, Cin, Z, Cout, Cnf1-Cnf2),
    adder([], Xs, Cout, Zs, Cnf2-Cnf3) .
adder([X|Xs], [], Cin, [Z|Zs], Cnf1-Cnf3) :-
    half_adder(X, Cin, Z, Cout, Cnf1-Cnf2),
    adder([], Xs, Cout, Zs, Cnf2-Cnf3) .
adder([X|Xs], [Y|Ys], C, [Z|Zs], Cnf1-Cnf3) :-
    full_adder(X, Y, C, Z, NextC, Cnf1-Cnf2),
    adder(Xs, Ys, NextC, Zs, Cnf2-Cnf3) .

half_adder(X, Y, Sum, Carry, Cnf1-Cnf3) :-
    xor(Sum, X, Y, Cnf1-Cnf2),
    and(Carry, X, Y, Cnf2-Cnf3) .

full_adder(X, Y, CarryIn, Sum, CarryOut, Cnf1-Cnf4) :-
    half_adder(A, Y, S1, C1, Cnf1-Cnf2),
    half_adder(CarryIn, S1, Sum, C2, Cnf2-Cnf3),
    or(Carry, C1, C2, Cnf3-Cnf4) .

%% Z == X or Y
or(X, Y, Z, [[Z, -X], [Z, -Y], [-Z, X, Y] | Cnf]-Cnf) .

%% Z == X and Y
and(X, Y, Z, [[Z, X, Y], [-Z, X], [-Z, Y] | Cnf]-Cnf) .

%% Z == X xor Y
xor(X, Y, Z, [[Z, -X, Y], [Z, X, -Y],
              [-Z, X, Y], [-Z, -X, -Y] | Cnf]-Cnf) .

%% Z == (X == Y)
eq(X, Y, Z, [[-Z, -X, Y], [-Z, X, -Y],
             [Z, X, Y], [Z, -X, -Y] | Cnf]-Cnf) .

```

Figure 1. An encoding for a ripple carry adder

of literals, or equivalently a conjunction of clauses. Each literal is a propositional variable p or its negation $\neg p$. A truth assignment is a mapping from propositional variables into $\{0, 1\}$.

Before we introduce the specifics of our encoding of the telecommunications feature subscription configuration problem to SAT let us define a language in which to describe these specifics. We portray the basic components of an encoding and their composition into more complex components as logic programs. So, an encoding is a program which generates a CNF. Predicates in the encoding specify the relation between the components and their encoding to CNF. Composition of components is obtained as composition of programs and conjunction of CNFs.

We represent: literals as (Prolog) terms of the form X or $\neg X$ where X is a logic variable; clauses as lists of literals; and conjunctions of clauses as lists of clauses. A truth assignment is a substitution of the variables to the constants $\{0, 1\}$. We will represent binary numbers as lists of bits with the *most significant bit last*, instead of the usual notation which has it first. Hence $[1,0,1,1]$ represents the decimal number 13 or the binary number 1101. This choice facilitates the handling of addition where least significant bits are processed first. Encodings for binary adders play a major role in the application we describe in this paper, hence we illustrate the concept of declarative encoding with an encoding of binary addition given as Figure 1.

Ripple carry adder

Figure 1 illustrates the encoding of a textbook style (Cormen et al. 1990) ripple-carry circuit for binary addition. We will assume that

the lengths of the addends differ by at most one (with the extra bit on the second addend). The predicates `and/4`, `or/4` and `xor/4` describe the encoding of the basic gates with the fourth argument specifying the CNF as a difference list. The predicate `eq/4` is a gate which expresses that its inputs X and Y must be equal. For example, a call to `?- and(X, Y, Z, Cnf-[])` will bind the variable `Cnf` to a conjunctive normal form of the propositional formula $Z \leftrightarrow X \wedge Y$. The predicates `half_adder/5` and `full_adder/6` specify the adder gates and illustrate the composition of encodings. The goal `adder(Xs, Ys, C, Zs, Cnf-[])` describes the encoding (`Cnf`) of a circuit for ripple carry addition with inputs Xs and Ys with an additional carry-in bit C . The output of the circuit is the binary number Zs . A call to this predicate generates the circuit as a propositional formula in conjunctive normal form. For example, a call to

```
?- adder([X1,X2,X3,X4], [Y1,Y2,Y3,Y4], C, Zs, Cnf-[])
```

will bind the variable `Cnf` to a conjunctive normal form corresponding to the 4-bit binary adder with inputs $[X_1, X_2, X_3, X_4]$ $[Y_1, Y_2, Y_3, Y_4]$ and C and outputs Zs .

For encoding applications, such as that considered in this paper, we will typically express the encoding declaratively and then solve a problem instance by applying a SAT solver to its CNF encoding. In our experimentation we apply the MiniSat SAT solver (Eén and Sörensson 2004; MiniSAT) through its Prolog interface described in (Codish et al. 2008c).

Just for fun, consider the Prolog query

```
?- adder(Xs, Xs, C, [1,0,1,1], Cnf-[]), sat(Cnf) .
Xs = [0, 1, 1], C = 1 .
```

```
?- adder(Xs, Xs, C, [0,1,1], Cnf-[]), sat(Cnf) .
Xs = [1, 1], C = 0 .
```

which generates the propositional formula representing a circuit for addition and poses a query to the SAT solver: do there exist Xs and C such that $Xs+Xs+C=13$ (or such that $Xs+Xs+C=6$)? The predicate `sat/1` is part of the interface to MiniSat described in (Codish et al. 2008c).

When considering the design of efficient circuits, two major concerns are circuit size and circuit depth. For example a ripple-carry adder is linear in size and depth. For k -bits it consists of k full adders (size). The linear depth stems from the linear (“ripple”) propagation of the carry-in bit which eventually reaches the carry-out bit. In circuit theory the depth of a circuit is considered to be a measure of the time complexity of the circuit. This is the time it takes for an input signal to reach an output signal. When encoding a circuit to a propositional formula, the size and depth of the circuit also determine the size and depth of the formula. The depth of the circuit has an impact on the chains of propagation that can occur during SAT solving. Similar to the case for circuits, it is often beneficial to consider a larger circuit with a smaller depth. Although for SAT solving we are not measuring the time from input to output, the depth of a formula still influences the number of decisions that the solver must make to determine a value by unit propagation.

The literature is rich in papers from the 1960’s and onwards which present circuits for fast binary addition. We have experimented with encodings for many of these including: carry look ahead, parallel prefix adders, and conditional-sum adders, etc. Interestingly in our final version ripple carry adders appear preferable to the more complex adders with smaller depth. We hypothesize that this is because of two things: ripple carry adders use the least number of gates, just k full adders and introduce the least number of additional propositional variables $k - 1$ internal carries; and rip-

```

full_adder(X,Y,Cin,Sum,Cout,Cnf1-Cnf2) :-
    Cin == 0, !, half_adder(X,Y,Sum,Cout,Cnf1-Cnf2).
full_adder(X,Y,Cin,Sum,Cout,Cnf1-Cnf2) :-
    X == 0, !, half_adder(Cin,Y,Sum,Cout,Cnf1-Cnf2).
full_adder(X,Y,Cin,Sum,Cout,Cnf1-Cnf2) :-
    Y == 0, !, half_adder(X,Cin,Sum,Cout,Cnf1-Cnf2).

half_adder(X,Y,Sum,Cout,Cnf1-Cnf2) :-
    X == 0, !, Sum = Y, Cout = 0, Cnf1 = Cnf2.
half_adder(X,Y,Sum,Cout,Cnf1-Cnf2) :-
    Y == 0, !, Sum = X, Cout = 0, Cnf1 = Cnf2.

and(X,Y,Z,Cnf1-Cnf2) :-
    X == 0, !, Z = 0, Cnf1 = Cnf2.
and(X,Y,Z,Cnf1-Cnf2) :-
    Y == 0, !, Z = 0, Cnf1 = Cnf2.

xor(X,Y,Z,Cnf1-Cnf2) :-
    X == 0, !, Z = Y, Cnf1 = Cnf2.
xor(X,Y,Z,Cnf1-Cnf2) :-
    Y == 0, !, Z = X, Cnf1 = Cnf2.

```

Figure 2. Clauses for simplifying the resulting circuits produced.

ple carry adders give very short circuits from the most significant input bits to the most significant output bits (1 gate), and these are the most important decisions made in maximization.

Simplifying circuits

An advantage of the declarative encoding of the circuits is that it is easy to improve the resulting circuits. The code for `adder/6` in Figure 1 assumes a carry-in, but in many cases this is not required. By setting the carry-in to 0 we obtain a correct circuit but one which is not as small as required. Luckily the encoding is easy to improve so that we do not create overly complex circuits. We can add specialization code which takes advantage of fixed inputs to simplify the resulting circuit.

The code in Figure 2 illustrates how to create specialized circuits when one of the inputs is known to be 0. For example a `full_adder` with one input 0 becomes a `half_adder`, a `half_adder` simply copies the input as sum, and has carry-out 0, an `and` gate always outputs 0, and a `xor` just passes through the input as output. Adding these clauses before the original clauses for these predicates results in substantially simplified circuits. We can of course add simplifications for when some inputs are known to be 1, or when two inputs are known to be the same. All the declarative code making use of the circuit elements automatically benefits from the simplification.

4. Encoding Feature Subscription

This section presents the main components required to encode an instance of the telecommunications feature subscription configuration problem to a propositional statement so that a Boolean SAT solver can be applied to derive a solution for the given instance. In the following sections these components are discussed in more detail.

Let $S = \langle F, C, U, W_F, W_U \rangle$ be a subscription for a given catalog $\langle F_c, P_c \rangle$. We seek an optimal relaxation

$$S' = \langle F', C', U', W_{F'}, W_{U'} \rangle.$$

With each feature $f \in F$ we associate a propositional variable b_f indicating if f is included in F' . With each constraint $p \in C$ (and $p \in U$) we associate a propositional variable b_p to indicate if p is included in C' (or in U').

Encoding Catalog Constraints For each constraint $p = (f > g)$ in C , p is included in the relaxation C' if and only if both f and g are included in F' . Hence we introduce to the encoding the propositional constraint

$$b_p \leftrightarrow b_f \wedge b_g \quad (1)$$

Encoding User Constraints For each constraint $p = (f > g)$ in U , if p is included in the relaxation U' then both f and g must be included in F' . Hence we introduce to the encoding the propositional constraint

$$b_p \rightarrow b_f \wedge b_g \quad (2)$$

A user constraint p may be excluded from relaxation U' either by excluding one of f or g (setting b_f or b_g to false) or by excluding the constraint (setting only b_p to false).

Encoding Acyclicity An assignment of truth values to the propositional variables b_p determinethe subsets C' and U' of C and U in a relaxation. These subsets must be such that $C' \cup U'$ represent a partial order and in particular, contain no cycles. For each constraint $p = (f > g)$ in $C \cup U$ we introduce to the encoding the propositional constraint

$$b_p \rightarrow \llbracket f > g \rrbracket \quad (3)$$

which intuitively states that if p is included in the relaxation then f is greater than g in the partial order. We will defer the full description of the formula $\llbracket f > g \rrbracket$ until the next section. The definition of this formula will depend on how we choose to model the acyclicity constraints.

Encoding Sum of Weights and Integer Comparison A relaxation S' is associated with a value $Value(S')$ equal to the sum of the weights of the features in F' and the constraints in U' (as determined by the propositional variables b_f and b_p of the encoding). For a given integer value m , we need to encode to propositional logic the constraint $Value(S') \geq m$. Let us denote this encoding by

$$\sum_{f \in F} w_f * b_f + \sum_{p \in U} w_p * b_p \geq m \quad (4)$$

We explain how this constraint is encoded in Section 6.

Finding an Optimal Relaxation A SAT solver can only answer questions of satisfaction, not optimization, hence we need to determine how we will determine an optimal relaxation. Given an encoding based on the above equations (1), (2), (3) and (4) for fixed m we have a satisfaction problem. We can thus find the maximal value of m via a series of satisfaction questions. Possible approaches are: starting from a low value of m and increasing it until we find a value which has no solution, starting from a high value of m and decreasing it until we find a satisfiable value, or binary search over the values of m . We explore these possibilities in detail in Section 7.

5. Encoding Acyclicity

Consider the set of propositional variables b_p which correspond to the precedence constraints (catalog constraints, C and user preferences U). A truth assignment on these variables determines a relaxation ($C' \subseteq C$ and $U' \subseteq U$) which is required to be acyclic. Two main approaches to encode acyclicity are discussed in the literature. In the paper (Codish et al. 2008b) the authors term these: *atom-based* and *symbol-based*.

Atom-based encoding

The encoding of the telecommunications feature subscription configuration problem to SAT described in (Lesaint et al. 2008) is

atom-based. It models a constraint of the form $p = (f > g)$ (the propositional variable b_p as an atom). So in this encoding $\llbracket f > g \rrbracket$ is simply a new propositional variable. To capture the meaning of these atoms (as a precedence) the encoding introduces propositional statements to capture the axioms of partial orders which these atoms are subject to. For example, to encode that if $p_1 = (a > b)$ and $p_2 = (b > c)$ are true then also $p_3 = (a > c)$ is true, a clause $p_1 \wedge p_2 \rightarrow p_3$ is introduced, and to encode that at least one of $p_1 = (a > b)$ and $p'_1 = (b > a)$ can be true, a clause $\neg p_1 \vee \neg p'_1$ is introduced. For an instance involving n features, this encoding will introduce $O(n^2)$ propositional variables and involve $O(n^3)$ clauses to express acyclicity.

Symbol-based encoding (binary representation)

A symbol-based approach to encoding *partial order constraints* is introduced in (Codish et al. 2008b). Informally, a partial order constraint is just like a formula in propositional logic except that statements may involve propositional variables as well as atoms of the form $(f > g)$ where f and g are symbols.

In that paper the authors present an encoding in which each symbol is represented using $k = \lceil \log_2 n \rceil$ propositional variables which model the binary representation of its integer value. This same approach was experimented with for the encoding of telecommunications feature subscription configuration problems and initial results were reported in (Codish et al. 2008a). Each feature f is associated with an integer variable encoded as a binary number requiring $k = \lceil \log_2 n \rceil$ bits. The formula $\llbracket f > g \rrbracket$ is simply an encoding of the integer “greater than” relationship on the binary encodings of f and g .

For an instance involving n features, this requires $k = \lceil \log_2 n \rceil$ additional propositional variables for each feature. Constraints of the form illustrated in Equation (3) are then straightforward to encode in k -bit arithmetic and involve $O(\log n)$ clauses each. Overall the encoding requires a total of $O(n \log n)$ additional propositional variables and $O(k \log n)$ clauses, where $k = |C \cup U|$. Typically, k is much smaller than the worst case which is n^2 .

Figure 3 illustrates an encoding for a constraint of the form $B \leftrightarrow \llbracket f > g \rrbracket$. The goal

```
xs_gt_ys([F0, ..., Fk-1], [G0, ..., Gk-1], B, Cnf)
```

where $[F_0, \dots, F_{k-1}]$ and $[G_0, \dots, G_{k-1}]$ are the k -bit binary integers corresponding to features f and g respectively, binds the logic variable Cnf to the conjunctive normal (as a difference list) of the formula for $B \leftrightarrow \llbracket f > g \rrbracket$. Recall that binary numbers are represented as lists of bits, least significant first. The encoding is based on the specification that for the case when f and g are represented by more than a single bit then $f > g$ holds if either $f' > g'$, for the high bits f' and g' of f and g or else if $f' = g'$ and $f'' > g''$, for the low bits f'' and g'' of f and g .

The encoding problem described in (Codish et al. 2008b) is concerned with proving termination of term rewrite systems using lexicographic path orders (LPO). The experiments described in (Codish et al. 2008b) apply a partial order constraint solver written in SWI-Prolog (Wielemaker 2003) which interfaces the MiniSat solver (Eén and Sörensson 2004) for solving SAT instances as described in (Codish et al. 2008c). The results presented in (Codish et al. 2008b) indicate a clear advantage in the use of a symbol-based encoding over the use of an atom-based encoding for the LPO termination problem. For the preliminary results presented in ICLP08 (Codish et al. 2008a) we applied the same constraint solver. Experimental evaluation shows that for many of the instances of feature subscription the atom based encoding of (Lesaint et al. 2008) is superior to our symbol based encoding. This is particularly true for instances which have a high ratio of constraints per feature.

```
xs_gt_ys([X], [Y], B, Cnf1-Cnf2) :- !,
    gt(X, Y, B, Cnf1-Cnf2).
xs_gt_ys(Xs, Ys, B, Cnf1-Cnf6) :-
    split(Xs, LoXs, HiXs),
    split(Ys, LoYs, HiYs),
    xs_gt_ys(HiXs, HiYs, B1, Cnf1-Cnf2),
    xs_eq_ys(HiXs, HiYs, B2, Cnf2-Cnf3),
    xs_gt_ys(LoXs, LoYs, B3, Cnf3-Cnf4),
    and(B2, B3, Tmp, Cnf4-Cnf5),
    or(B1, Tmp, B, Cnf5-Cnf6).

xs_eq_ys([X], [Y], B, Cnf1-Cnf2) :- !,
    eq(X, Y, B, Cnf1-Cnf2).
xs_eq_ys([X|Xs], [Y|Ys], B, Cnf1-Cnf4) :- !,
    eq(X, Y, B1, Cnf1-Cnf2),
    xs_eq_ys(Xs, Ys, B2, Cnf1-Cnf4),
    and(B1, B2, B, Cnf3-Cnf4).

%% Z == X > Y (equivalently Z == X * -Y)
gt(X, Y, Z, [[Z, -X, Y], [-Z, X], [-Z, -Y] | Cnf] - Cnf).
```

Figure 3. Symbol-based encoding with binary number representation.

On the one hand the atom-based encoding of acyclicity is larger than the corresponding symbol-based encoding. On the other, it has much more potential to facilitate unit propagation. In each of the n^3 clauses of the form $p_1 \wedge p_2 \rightarrow p_3$ (for $p_1 = (a > b)$, $(p_2 = b > c)$ and $p_3 = (a > c)$) once the solver has fixed any two of the choices $p_1, p_2, \neg p_3$, then the third follows by unit propagation. In contrast in the symbol-based approach, fixing individual bits or even pairs of bits in the binary integer representation rarely determines other bits of the representation.

Symbol-based encoding (unary representation)

In this paper we present an alternative symbol based encoding which is significantly better (experimentally) than the two others. It also has superior potential for unit propagation.

The unary encoding of a non-negative integer in n bits is a sequence of n values v_1, v_2, \dots, v_n such that $v_1 \geq v_2 \geq \dots \geq v_n$. The value of such a representation is the number of bits taking value 1. For example the sequence 11100000 represents the number 3 in 8 bits. Figure 4 illustrates the symbol-based encoding of partial orders with unary numbers. The predicate `unary/2` is the encoding of a list of bits as a unary number. For each pair of consecutive variables x_i, x_{i+1} a clause $x_{i+1} \rightarrow x_i$ is introduced to the encoding. The predicate `xs_gt_ys_unary/4` is the encoding of a constraint of the form $\llbracket f > g \rrbracket$ by the goal

```
xs_gt_ys_unary([F0, ..., Fn-1], [G0, ..., Gn-1], B, Cnf)
```

where $[F_0, \dots, F_{n-1}]$ and $[G_0, \dots, G_{n-1}]$ are the n -bit unary integers corresponding to symbols f and g respectively. The goal binds the variable Cnf to the conjunctive normal form of the formula $B \leftrightarrow f > g$.

The clauses encode $f > g$ if and only if $f-1 \geq g$ where $f-1$ is obtained by removing the first bit of f which must be a 1. Encoding that $f \geq g$ boils down to the requirement on the bits at each position i in the representation: $g_i \rightarrow f_i$ (if the g bit is a 1 then so must be the f bit. In this context the representation of g has one bit more than that of f (as one bit of f is already removed).

For an instance involving n features, this encoding introduces $O(n^2)$ propositional variables (n per symbol) and involves $O(kn)$ clauses (n per constraint) to express acyclicity, where $k = |C \cup P|$. In principle it is similar in size compared to the atom-based

```

unary([_], Cnf-Cnf) :- !.
unary([X1,X2|Xs], [[X1,-X2]|Cnf1]-Cnf2) :-
    unary([X2|Xs], Cnf1-Cnf2).

% [X|Xs] > Ys iff X and Xs >= Ys
xs_gt_ys_unary([X|Xs], Ys, B, Cnf1-Cnf3) :-
    xs_ge_ys_unary(Xs, Ys, R, Cnf1-Cnf2),
    and(X, R, B, Cnf2-Cnf3).

x_ge_y_unary([], [Y], B, [[-B,-Y], [B,Y]|Cnf]-Cnf).
x_ge_y_unary([X|Xs], [Y|Ys], B, [[-B,-X,Y], [-B,R],
    [B,-X,-R], [B,Y,-R]|Cnf1]-Cnf2) :-
    x_ge_y_unary(Xs, Ys, R, Cnf1-Cnf2).

```

Figure 4. Symbol-based encoding with unary number representation

encoding. However, the potential for propagation is enormous. Any single bit in the representation of a symbol f which is assigned any value, determines by unit propagation several additional symbols in the encoding of each precedence constraint involving f . Consider a constraint $f > g$ and denote the bits of f and g as f_0, \dots, f_{n-1} and g_0, \dots, g_{n-1} . If a single bit f_i is set to 1, then all of the bits f_0, \dots, f_{i-1} are determined by unit propagation and set to 1. If f_i is set to 0 then all of f_{i+1}, \dots, f_{n-1} are determined by unit propagation and set to 0. The other case, where g_i is set, is similar, but in this case the bits in g and in f are set.

The experimental evaluation presented in Section 8 illustrates the advantage of this encoding which is much faster than the others.

6. Encoding Weighted Sums

The decision problem associated with the telecommunications feature subscription configuration problem is: *given a parameter m is there a solution S' with value(S') $\geq m$.* To encode this we need a circuit to sum the weights in order to compare with the parameter m .

Unweighted Sum

We shall start by discussing how to encode the sum of unweighted bits in a set Q : $\sum_{p \in Q} p$, where $q = |Q|$.

We first present a simple divide and conquer approach. To sum q bits we introduce encodings for $q/2$ adders: 1 $q/2$ -bit adder, 2 $q/4$ -bit adders, ..., $q/2$ 1-bit adders. So the number of adders is linear and the depth of the deepest adder is $\log q$.

The Prolog code in Figure 5 illustrates this process of summing a set of q bits. A call to `sum(Bits, Sum, Cnf-[])` generates a binary integer representation `Sum` of the list of `Bits` together with the conjunctive normal form, `Cnf`, which specifies the relation between `Bits` and `Sum`. A ripple carry adder (predicate `adder/5`) was presented in Figure 1. Remember that the third argument is the carry-in which can be utilized here to take one of the original q bits. The call to `adder(Sum1, Sum2, B1, Sum, Cnf3-Cnf4)` constructs the conjunctive normal form for the binary `Sum` of numbers `Sum1` and `Sum2` with the carry-in `B1`.

This encoding introduces $O(q)$ additional propositional variables and $O(q)$ clauses, but the longest chain of gates in the circuit is $O((\log q)^2)$.

Consider the sum of 11 bits. This is split into two 5-bit sums with the remaining bit being a carry-in. Each 5-bit sum is split into two 2-bit sums and a carry-in bit. The circuit created is shown in Figure 6. The width of the line indicates the worth of the bit. A carry-in to the 2-bit and 3-bit adders is shown entering the left hand side.

```

sum([], [], Cnf-Cnf).
sum([X], [X], Cnf-Cnf).
sum([X,Y], [S,C], Cnf1-Cnf2),
    half_adder(X,Y,S,C,Cnf1-Cnf2).
sum_cnf([B1,B2,B3], [S,C], Cnf1-Cnf2) :-
    full_adder(B1,B2,B3,S,C,Cnf1-Cnf2).
sum([B1,B2,B3|Bs], Sum, Cnf1-Cnf4) :-
    split([B2,B3|Bs], Xs, Ys),
    sum(Xs, Sum1, Cnf1-Cnf2),
    sum(Ys, Sum2, Cnf2-Cnf3),
    adder(Sum1, Sum2, B1, Sum, Cnf3-Cnf4).

```

Figure 5. Summing bits by divide and add

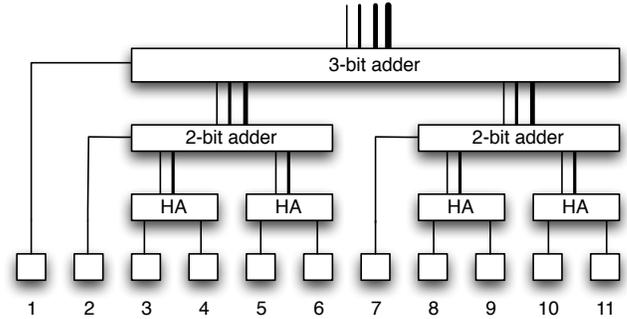


Figure 6. Circuit for summing 11 bits by divide and add.

Consider the application of this approach to sum 15 bits. In principle, this could be done by swapping each of the four half adders in Figure 6 with full adders, each one taking one of the additional four bits. In practice, this is exactly what happens. When the recursive call for `sum/3` (the last clause in Figure 5) receives exactly three bits `[X,Y,Z]` the subsequent call to `adder/5` is of the form `adder([X],[Y],Z,Sum,Cnf)` which introduces the full adder.

An alternative approach to summing bits associates a weight with each bit. Initially all bits have weight 1, and then iteratively, three bits with equal weights (w) are replaced by two bits: one with the same weight and the other doubled (w and $2w$). This process terminates when there are no longer 3 bits with equal weights. At this stage the remaining bits can be viewed as belonging to a pair of binary numbers. The transformation of three to two bits is performed with a full adder, and hence we term these two bits: a sum and a carry bit. An encoding of this approach is presented in Figure 7.

The call to predicate `sum(Bits, Sum, Cnf1-Cnf3)` reduces the elements of `Bits` into two binary numbers `Num1` and `Num2` the `Sum` of which is provided by a binary adder. The heart of the code is `iterate_3x2` which given a list of bits, calls `once_3x2` to repeatedly take each 3 bits from the list and create a sum and a carry bit using a full adder. This process creates a set of sum bits and a set of carry bits. The set of sum bits are passed into a recursive call to `iterate_3x2` to reduce the original bits to a pair of sum bits and a list of carry bits. The code for `reduce` uses `iterate_3x2` to create the two lowest bits of the numbers `Num1` and `Num2`, and then recursively reduces the accumulated carry bits to determine the other bits of `Num1` and `Num2`. Finally, adding these two binary numbers gives the sum of the original bits.

The `iterate_3x2` reduction process on q bits introduces at most $\lceil q/3 \rceil + \lceil q/9 \rceil + \lceil q/27 \rceil + \dots$ additional propositional

```

sum_cnf(Xs, Sum, Cnf1-Cnf3) :-
    reduce(Xs, Num1, Num2, Cnf1-Cnf2),
    add_cnf(Num1, Num2, Sum, Cnf2-Cnf3).

reduce([], [], [], Cnf-Cnf) :- !.
reduce(Xs, [X|Num1], [Y|Num2], Cnf1-Cnf3) :-
    iterate_3x2(Xs, [X, Y], Cs-[], Cnf1-Cnf2),
    reduce(Cs, Num1, Num2, Cnf2-Cnf3).

iterate_3x2([], [0,0], Cs-Cs, Cnf-Cnf) :- !.
iterate_3x2([X], [X,0], Cs-Cs, Cnf-Cnf) :- !.
iterate_3x2([X, Y], [X, Y], Cs-Cs, Cnf-Cnf) :- !.
iterate_3x2(Xs, [X, Y], Cs1-Cs3, Cnf1-Cnf3) :-
    once_3x2(Xs, Ss, Cs1-Cs2, Cnf1-Cnf2),
    iterate_3x2(Ss, [X, Y], Cs2-Cs3, Cnf2-Cnf3).

once_3x2([X, Y, Z|Xs], [S|Ss],
         [C|Cs1]-Cs2, Cnf1-Cnf3) :- !,
    full_adder(X, Y, Z, S, C, Cnf1-Cnf2),
    once_3x2(Xs, Ss, Cs1-Cs2, Cnf2-Cnf3).
once_3x2(Xs, Xs, Cs-Cs, Cnf-Cnf).

```

Figure 7. Summing bits three by two

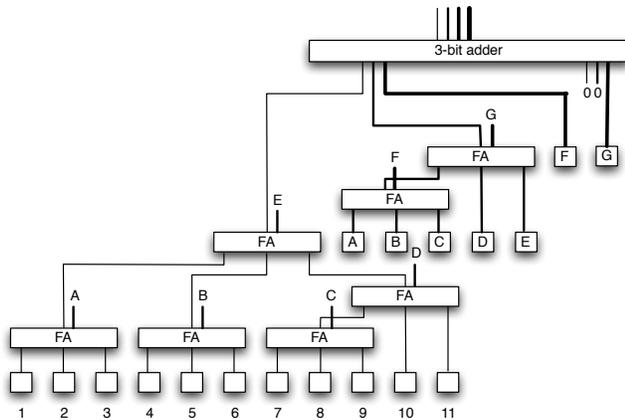


Figure 8. Circuit for summing 11 bits by three by two.

variables (which is $O(q)$), and $O(q)$ clauses with a maximal depth of $O(\log_3 q)$. Overall the reduction process adds $O(q)$ additional propositional variables and $O(q)$ clauses with a maximum depth of $O(\log_3 q \cdot \log_2 q)$. The final addition adds at most $O(\log_2 q)$ variables and $O(\log_2 q)$ clauses.

This encoding also has the advantage that it never uses more bits than necessary to encode intermediate values, since they are always lists of bits. Consider again the problem of adding 11 bits. The circuit resulting from summing three by two is shown in Figure 8. For the full adders the carry bit is named and assumed connected to the same named input box. Overall the three by two approach requires 7 full adders and a 3-bit adder. Using the simplifications of Figure 2 discussed at the end of Section 3 the 3-bit adder is encoded as just a single half-adder. Compare this with the divide and add approach which requires 7 full adders and 4 half adders. Note also that the 11 bit sum is a case of a perfect balanced split in divide and add approach.

In our encoding we combine the two approaches presented here for summing bits. We encode the summing of a list *Bits* in two phases. First we apply the divide and add technique of Figure 5

```

sum_cnf(Bits1,Bits2,Bits3,Bits4,Sum,C1-C9) :-
    sum(Bits1,Sum1,C1-C2),
    sum(Bits2,Sum2,C2-C3),
    sum(Bits3,Sum3,C3-C4),
    sum(Bits4,Sum4,C4-C5),
    adder(Sum1, [0|Sum2], 0, Sum12, C5-C6),
    adder(Sum3, [0|Sum3], 0, Sum3x3, C6-C7),
    adder(Sum12, Sum3x3, 0, Sum123, C7-C8),
    adder(Sum123, [0,0|Sum4], 0, Sum, C8-C9).

```

Figure 9. A weighted sum for bits with weightings from 1 to 4.

to obtain from each 15 bits from the list *Bits* a four bit binary number. This results in a list *Nums* of four bit numbers. To this list we apply a three by two reduction process similar to that described in Figure 7, except that instead of combining 3 bits of weight w to obtain two bits of weights w and $2w$, we are combining three 4-bit numbers of weight w to obtain 2 numbers, one of weight w and one of weight $2w$. Experimentation indicates that this combination is considerably faster than each of the individual techniques.

Weighted Sum

We now discuss how to encode the sum of weighted bits in a set Q : $\sum_{p \in Q} w_p * p$. There are many approaches to determining weighted sums in Boolean circuits. For the problems of interest to us, the weights w_p range only over 1..4. Hence the straightforward approach is to simply sum up all the bits of the same weight (an unweighted sum) and then multiply each sum by its weight and add them. This is exemplified in the code of Figure 9.

The `sum_cnf` predicate shown in Figure 9 receives as inputs the lists of selection bits *Bits1*, *Bits2*, *Bits3*, *Bits4* corresponding to the weights, 1–4, and generates the formula as a conjunctive normal form (difference list *C1-C9*) which specifies the bits, *Sum*, of the weighted sum. Each individual set of bits is first summed, and the resulting binary number is then multiplied by its coefficient, in the cases of 2 and 4 by adding left shifts, and for 3 by adding twice the number to the number itself. The weighted sums are then added.

7. Maximization using SAT

In this section we discuss several algorithms and implementations which facilitate a SAT solver to address an optimization problem. Given a conjunctive normal form, *Cnf*, and a vector *Vec* of variables, we seek a satisfying assignment for *Cnf* which maximizes (or minimizes) the binary number represented by *Vec*. The Prolog interface to MiniSat described in (Codish et al. 2008c) provides this facility by means of a call to the predicate `maximize(Vec, Cnf)`. In (Codish et al. 2008c), the authors present an implementation which iterates on the bits in *Vec* (from most to least significant). In each iteration the next most significant bit of *Vec* is determined, if possible to a “1”, and otherwise, if this results in an unsatisfiable *Cnf* formula, then to a “0”. In each iteration we keep only the bindings to the variables in *Vec* and forget any other binding from previous calls to the SAT solver. For a vector *Vec* with k bits, this requires at most k calls to the SAT solver. Sometimes less, as perhaps by chance some of the next bits to be considered were already set to “1”. If *Vec* represents the sum of n bits then this involves $\log n$ calls to the SAT solver. The code is depicted in Figure 10.

Note that the call “`\+ (B=1, solve(Cnf))`” succeeds if the *Cnf* is unsatisfiable when $B=1$, but due to the use of negation it does not bind the variables. To simplify presentation, the code in Figure 10 instantiates only the variables that appear in *Vec* and no other variables in *Cnf*. This is easily rectified if the satisfying assignment is required.

```

maximize([],_Cnf) :-
    \+ \+ solve(Cnf).
maximize([B|Bs],Cnf) :-
    maximize(Bs,Cnf),
    ( \+ (B=1, solve(Cnf)) -> B=0 ; B=1 ).

```

Figure 10. Maximization by binary search.

```

maximize(Vec,Cnf) :-
    solve(Cnf), assert(soln(Vec)), fail.
maximize(Vec,Cnf) :-
    soln(_),
    improve(Vec, Cnf).

improve(Vec,Cnf) :-
    retract(soln(Last)),
    xs_gt_ys(Vec,Last,Cnf1-Cnf),
    (solve(Cnf1) ->
        assert(soln(Vec)), fail
    ;
        Vec = Last
    ),!.
improve(Vec, Cnf) :-
    improve(Vec, Cnf).

```

Figure 11. Maximization by linear search.

An advantage of the approach described in Figure 10 is that the number of calls to the SAT solver is bounded by the length of `Vec`. A disadvantage is that, typically, the call to a SAT solver for an unsatisfiable formula is far more expensive than that for a satisfiable formula. Hence each “0” bit in the maximal value of `Vec` is more costly to obtain. This approach is termed “binary search” because each step eliminates half of the search space by setting the next most significant bit.

Figure 11 presents an alternative approach which, starting from any initial value of `Vec` for which `Cnf` is satisfiable, repeatedly seeks to increase that value and still satisfy `Cnf`. Once the current value of `Vec` can not be increased (and still satisfy `Cnf`), a maximum has been reached.

The first clause for `maximize/2` generates the initial solution, records (`assert/1`) it, and then fails in order to unbind the variables and initiate the “improvement” by a failure-driven loop invoked by the second clause of `maximize/2` and specified as predicate `improve/2`. Note that the second clause of `maximize/2` calls `improve/2`, only if it finds a current solution, which means that the formula `Cnf` is satisfiable, otherwise the call to `maximize/2` fails.

A call of the form `improve(Vec,Cnf)` receives a formula, `Cnf` and a vector, `Vec` and aims to (repeatedly) bind `Vec` to a number larger than the previous value of `Vec`. In the first clause for `improve/2`, the previous value of `Vec` is picked up and called `Last` by the call to `retract/1`. If this `retract` fails then the original `Cnf` was not satisfiable (the call to `solve(Cnf)` in `maximize` failed) and failure is reported. Otherwise, the call to `xs_gt_ys/3` adds to the `Cnf` the clauses which encode that `Vec` must be larger than `Last`. The resulting `Cnf1` is provided to the SAT solver. If `Cnf1` is satisfiable, then the new value of `Vec` is recorded (`assert/1`) and the call to `fail` invokes iteration, by means of the second clause of `improve/2` first unbinding all assignments in the `Cnf`. If otherwise `Cnf1` is not satisfiable, then the value in `Last` is the maximum, and the iteration terminates.

The advantage of this approach is that all calls to the SAT solver, except the last one, are satisfiable calls. A disadvantage is that the

number of calls to the SAT solver may be exponential in the length of `Vec`. If `Vec` represents the sum of n bits then maximization may require n calls to the SAT solver. However, experimental evidence indicates that the number of calls to the SAT solver is typically much less. This approach is termed “linear search” because the search proceeds linearly.

For the optimization algorithm described in Figure 11, each time the underlying SAT solver is re-invoked, it is considering a formula which is more restricted than in the previous invocation. Clauses further restricting the value of `Vec` have been added to the formula of the previous invocation. This means that the conflict clauses learned by the underlying sat solver (in this case MiniSat) in one call are still valid for the subsequent calls.

The code depicted in both Figures 10 and 11 restarts the underlying SAT solver from scratch with each invocation and as such all conflict clauses are flushed. By extending the Prolog interface to the underlying SAT solver, we can control which learned conflict clauses are kept and we can incrementally add clauses. This enables to considerably improve the optimization techniques presented in Figures 10 and 11.

To apply the new interface, first the predicate `sat_init/0` should be called in order to initialize the solver. Then any of the following operations can be taken at any time:

- `sat_add_clauses(Cnf,Vs,SatVs)`: it adds to the solver’s data-base the clauses in `Cnf`. Note that it assigns to every logic variable in `Cnf` a corresponding *sat variable* that must be used in further references to such a variable. In order to get the *sat variable* that is assigned to each logic variable, the user should provide a list of logic variables in the second argument `Vs` and the predicate will bind `SatVs` to a list of the corresponding *sat variables*. A *sat variable* is an integer value (negative means negated);
- `sat_solve(Assumes)`: it succeeds if there exists an assignment that satisfies the current formula. Namely, all clauses added by all calls to `sat_add_clauses/3` since the last call to `sat_init/0` and under the assumptions in the list `Assumes`. An assumption is a literal (of the form `X` or `-X` where `X` is a *sat variable*) which is assumed to be true. Namely, the variable `X` is assumed to be *true* or *false*, respectively. Note that the conflict clauses learned during this query are not cleared on exit, but are maintained in order to use them in further calls to the predicate. The assumptions provide a mechanism for checking if there exists a satisfying assignment that satisfies some conditions, without the need to add clauses that force the condition, since we cannot later undo the addition of such clauses without clearing the learned conflict causes.
- `sat_get_values(SatVs,Values)`: given a list of *sat variables* `SatVs`, this predicate binds `Values` to the list of values (0 and 1) assigned to those *sat variables* in the satisfying assignment that was found after the last call to `sat_solve/1`.

At the end, the predicate `sat_deinit` should be called in order to de-initialize the underlying solver.

The code in Figure 12 implements the binary search algorithm of Figure 10 using the new interface. It maintains all of the learned conflict clauses for use in subsequent invocations of the SAT solver. It processes the bits from most to least significant, therefore in the first line it reverses the list `Vec` to `Vec_MSB` for convenience. In the next lines: it initializes the SAT solver; adds the clauses of `Cnf` to the solver’s data-base and gets back a list of *sat variables* `Vec_MSB_SVars` which corresponds to the list of logic variables `Vec_MSB`; calls the underlying SAT solver to check if the formula is satisfiable; then `set_one_prefix/4` extracts the maximum prefix of *sat variables* from `Vec_MSB_SVars` that are assigned to 1 in the

```

maximize(Vec,Cnf) :-
    reverse(Vec,Vec_MSB),
    sat_init,
    sat_add_clauses(Cnf,Vec_MSB,Vec_MSB_SVs),
    sat_solve([]),
    set_one_prefix(Vec_MSB_SVs,Vec_MSB,SVs_1,Vs_1)
    maximize_loop(SVs_1,Vs_1),
    sat_deinit, !.

maximize_loop([],[]).
maximize_loop([SV|SVs],[V|Vs]) :-
    ( sat_solve([SV]) ->
      set_one_prefix([SV|SVs],[V|Vs],SVs_1,Vs_1)
    ;
      sat_add_clauses([-SV],_,_),
      SVs_1=SVs,
      Vs_1=Vs,
    ),
    maximize_loop(SVs_1,Vs_1).

set_one_prefix([SV|SVs],[V|Vs],SVs_1,Vs_1) :-
    sat_get_values([SV],[V]),
    V = 1,
    sat_add_clauses([SV],_,_), !,
    set_one_prefix(SVs,Vs,SVs_1,Vs_).
set_one_prefix(SVs,Vs,SVs,Vs).

```

Figure 12. Maximization by binary search maintaining conflict clauses.

last call and adds clauses to force them to be one in further calls to the solver (also binds the corresponding logic variables to 1); calls `maximize_loop/2` which on exit guarantees that `Vec_MSB` is bound to the maximum value; and the last line de-initializes the solver.

The predicate `maximize_loop/2` has two arguments, the first one is a list of sat variables to be maximized and the second is a list of the corresponding logic variables. In each iteration it asks the underlying SAT solver if there exists a satisfying assignment under the *assumption* that the next bit `SV` is 1 (the call `sat_solve([SV])`), if yes then `set_one_prefix/4` extracts the maximum prefix of `[SV|SVs]` that takes values 1 in that assignment and adds clauses that force them to be 1 in the next iterations (and binds the corresponding variables in `[V|Vs]` to 1). Otherwise, it adds a clause that forces `SV` to be 0 in the next iterations. This loop is repeated until all bits are processed. This code still might require $\log n$ iterations, but it has two advantages over the one of Figure 10: it takes advantage of the learned conflict clauses; and it eliminates the prefix which is already known to be 1 in the current assignment.

The code in Figure 13 implements the linear search algorithm of Figure 11 using the new interface. It maintains all of the learned conflict clauses for use in subsequent invocations of the SAT solver. In the first clause: it initializes the sat solver; adds the clauses of `Cnf` to the solver's data-base and gets back a list of sat variables `Vec_SVs` which corresponds to the list of logic variables `Vec`; calls the sat solver and gets the initial maximum `Curr_Max`; calls `improve/3` (with the current maximum) which on exit guarantees that `Vec` is bound to the maximum value; and the last line de-initializes the solver. The predicate `improve/3` has three arguments, the first one is a list of sat variables to be maximized, the second corresponds to the last maximum, and the last argument is used to return the actual maximum in the last iteration. In each iteration of `improve/3`: it adds to the solver's data-base clauses which force the variables of `Vec` to be greater than the

```

maximize(Vec,Cnf) :-
    sat_init,
    sat_add_clauses(Cnf,Vec,Vec_SVs),
    sat_solve([]),
    sat_get_values(Vec_SVs,Curr_Max),
    improve(Vec_SVs,Curr_Max,Vec),
    sat_deinit, !.

improve(Vec,Last_Max,Final_Max) :-
    xs_gt_ys(Vec,Last_Max,Cnf-[]),
    sat_add_clauses(Cnf,_,_),
    sat_solve([]),
    sat_get_values(Vec,Curr_Max), !,
    improve(Vec,Curr_Max,Final_Max).
improve(_Vec,Final_Max,Final_Max).

```

Figure 13. Maximization by linear search maintaining conflict clauses.

last maximum `Last_Max`; then if there exists a satisfying assignment to the new formula it continues considering `Curr_Max` as the last maximum. This loop is repeated until the call `sat_solve([])` fails, which indicates that `Last_Max` is the actual maximum, then the second clause of `improve/3` is activated and binds the third argument to the actual maximum.

8. Experimental results

In this section we show the effect of various encodings on the speed of the resulting SAT solution. The experimentation is based on a collection of catalogs and feature subscriptions created by the authors of (Lesaint et al. 2008). So the tables now compare the same instances. This was not the case for the tables presented in (Codish et al. 2008a). All together there are 270 instances, 10 per configuration. The 40 smallest configurations are not detailed in the tables.

First, we present a comparison with the published results from (Lesaint et al. 2008). The numbers for the SAT encoding described in (Lesaint et al. 2008) have improved considerably in the last year due to extensive work on the Sat4J solver. We are not aware of any improvements for the constraint programming and integer linear programming based solutions.

Table 1 describes the experiments for catalogs with 50 features and 250 precedence constraints (involving $\{<, >\}$). Each row labeled by $\langle f, p \rangle$ specifies a subscription with f features and p user precedence constraints with weights selected between 1 and 4. Times are measured in seconds and averaged for each set of 10 instances. The combined column marked `pocsp (U/B)` corresponds to our Prolog implementation of partial order constraints with unary and binary treatment of partial order and built on top of MiniSat (average times over 10 random instances¹). The columns marked `pwmsat`, `cp1ex` and `cp` are the times taken from Table 2 of (Lesaint et al. 2008) for their: SAT encoding², ILP solver and CP solver. The machines are also different. There is a PC Pentium 4 (CPU 1.8 GHz and 768 MB of RAM). Ours is a PC Pentium 4 (CPU 2.4 GHz and 512 MB of RAM). Ours is running SWI Prolog under Linux, kernel 2.6.11-6. With no intention to compare the two machines, the timings are clear enough.

Tables 2 and 3 illustrate results for larger catalogues $\langle 50, 500, \{<, >, <>\} \rangle$ (50 features and 500 precedence con-

¹The precise instances used may be found at http://www.cs.bgu.ac.il/~mcodish/Papers/Pages/feature_subscription.html

²The authors of (Lesaint et al. 2008) use the SAT4J solver - <http://www.sat4j.org/>.

$\langle f, p \rangle$	pocsp (U/B)		pwmsat	cplex	cp
$\langle 30, 20 \rangle$	0.10	0.20	6.40	1.02	0.65
$\langle 35, 35 \rangle$	0.37	1.58	23.95	22.76	7.43
$\langle 40, 40 \rangle$	1.31	5.97	282.76	247.14	67.80
$\langle 45, 90 \rangle$	18.72	266.11	12638.25	7690.90	1115.51
$\langle 50, 4 \rangle$	1.58	9.52	195.72	1010.38	413.61

Table 1. Timings (sec) for catalog $\langle 50, 250, \{<, >\} \rangle$

$\langle f, p \rangle$	pocsp (U/B)		pwmsat	cplex	cp
$\langle 30, 20 \rangle$	0.01	0.01	4.09	0.48	0.42
$\langle 35, 35 \rangle$	0.36	1.04	6.84	1.82	1.64
$\langle 40, 40 \rangle$	0.54	2.28	11.31	3.02	3.91
$\langle 45, 90 \rangle$	1.57	7.08	59.27	17.45	14.80
$\langle 50, 4 \rangle$	0.96	3.87	21.47	3.77	16.92

Table 2. Timings (sec) for catalog $\langle 50, 500, \{<, >, <>\} \rangle$

$\langle f, p \rangle$	pocsp (U/B)		pwmsat	cplex	cp
$\langle 30, 20 \rangle$	0.01	0.01	5.03	18.46	2.38
$\langle 35, 35 \rangle$	0.04	0.10	18.28	126.35	12.88
$\langle 40, 40 \rangle$	3.14	94.69	92.11	514.27	42.27
$\langle 45, 90 \rangle$	14.19	914.58	2443.23	3780.54	188.83
$\langle 50, 4 \rangle$	10.72	483.33	319.53	3162.08	342.49

Table 3. Timings (sec) for catalog $\langle 50, 750, \{<, >\} \rangle$

straints involving constraints from $\{<, >, <>\}$ and $\langle 50, 750, \{<, >\} \rangle$ (50 features and 750 precedence constraints involving constraints from $\{<, >\}$). Once again weights are values between 1 and 4 and times are measured in seconds. The columns marked *pwmsat*, *cplex*, and *cp* are the times taken from Table 3 of (Lesaint et al. 2008).

Table 4 provides a more recent comparison between the atom-based and symbol-based encodings to SAT and contains five columns of results for each configuration. The bottom 2 rows in the table, labeled *Total* and *Max*, contains the total and maximal values for all 270 instances. The first two columns of results, *pocsp* (U/B), are identical to the corresponding columns in Tables 1-3. They present the timings for our Prolog implementation of partial order constraints using unary and binary representations built on top of MiniSat. The next three columns *pwmsat* (U/B/At) present timings (sec) obtained using the partial weighted MAXSAT solver of Sat4J for three encodings: our symbol based approach using unary and binary representation and the atom based encoding of (Lesaint et al. 2008). The numbers in these three columns were obtained on the same machine and using the same configuration of the SAT solver. The machine is a Pentium IV 3GHz, 32 bits architecture running Linux 2.6.29.1 and Java 1.6.0_13 (and provided by Daniel Le Berre).

We first note that there is a significant improvement in the numbers for *pwmsat* observable by comparing the rightmost column of Table 4 with the corresponding *pwmsat* columns in Tables 1—3. This improvement represents an improvement in the *pwmsat* solver of Sat4J (the atom-based encoding is the same).

Columns *pocsp*(B) and *pwmsat*(B) represent timings for the same encodings (symbol-based, binary representation) but are not directly comparable as they run on different machines using different techniques to reach the optimal solution. Likewise columns *pocsp*(U) and *pwmsat*(U) for the unary symbols based encoding.

Comparing the leftmost column (our symbol based encoding with unary representation running on our *pocsp* solver) and the rightmost column (the atom based encoding described in (Le-

catalog & $\langle f, p \rangle$	pocsp (U/B)		pwmsat (U/B/At)			
$\langle 50, 250 \rangle$	$\langle 30, 20 \rangle$	0.10	0.20	0.55	1.22	0.58
	$\langle 35, 35 \rangle$	0.37	1.58	2.37	7.12	1.40
	$\langle 40, 40 \rangle$	1.31	5.97	8.90	21.03	9.20
	$\langle 45, 90 \rangle$	18.72	266.11	178.40	1844.01	484.16
	$\langle 50, 4 \rangle$	1.58	9.52	6.69	11.97	30.72
$\langle 50, 500 \rangle$	$\langle 30, 20 \rangle$	0.01	0.01	0.12	0.15	0.07
	$\langle 35, 35 \rangle$	0.36	1.04	1.88	3.35	0.57
	$\langle 40, 40 \rangle$	0.54	2.28	2.61	5.31	0.91
	$\langle 45, 90 \rangle$	1.57	7.08	9.23	22.11	2.34
	$\langle 50, 4 \rangle$	0.96	3.87	4.52	8.77	2.39
$\langle 50, 750 \rangle$	$\langle 30, 20 \rangle$	0.01	0.01	0.12	0.16	0.07
	$\langle 35, 35 \rangle$	0.04	0.10	0.30	0.47	0.14
	$\langle 40, 40 \rangle$	3.14	94.69	16.91	153.75	3.22
	$\langle 45, 90 \rangle$	14.19	914.58	82.01	1205.12	24.64
	$\langle 50, 4 \rangle$	10.72	483.33	54.78	618.66	61.57
Total	561	18240	3833	39748	6259	
Max	61	1667	662	14115	1867	

Table 4. another comparison

learning	binary	linear	technique	time
off	2073	1912	d&a	907
on	1012	561	r3×2	1702
			combined	561

Table 5. Comparing the effect of search and learning (left); and comparing the effect of how bits are summed (right)

saint et al. 2008) running on the MaxSAT solver of Sat4J, we note that the total time for the symbol based approach is 10 times faster than that for the atom-based, and that the hardest instance takes 30 times longer to solve in the atom based approach. Note that in both cases this is the same instance, the one called *ncf50-ncp250-nuf45-nup90-mw4-ind8*.

Table 5 provides (left side) a comparison of the linear and binary search techniques for maximization, with and without conflict clause learning. The table indicates that without clause learning, linear and binary search have similar performance. The effects of clause learning are considerably more significant for the linear approach. The right side of Table 5 provides a comparison of three techniques for summing bits. The *d&a* technique is the one described in Figure 6. The *r3×2* technique is the one described in Figure 7. The *combined* technique is the one that first applies *d&a* to derive 4-bit numbers and then applies to these the *r3×2* technique. On both sides of the table times are in seconds and denote the full time (encoding and SAT solving) for the entire benchmark suite of 270 instances using the *pocsp* approach with unary encoding. The Prolog code that appear in this paper can be downloaded from http://www.cs.bgu.ac.il/~mcodish/Software/ppdp09_code.pl. The code of the maximization predicates (Figures 12 and 13) together with the corresponding MiniSat Prolog interface can be downloaded from <http://www.cs.bgu.ac.il/~mcodish/Software/swi-minisat2.zip>.

9. Conclusion

Modeling a combinatorial optimization problem in SAT can be a significant challenge, when it contains features such as integers and ordering and optimization. There are many choices for modeling each component of the problem. By using a declarative language to encode modeling choices we are able to simplify experimentation with different approaches. We can also easily create simplified versions of the circuits that take into account knowledge that is fixed at

circuit creation time. We have applied our declarative programming encodings to the telecommunications feature subscription problem, creating a solution considerably better than previous published approaches. We believe there is great scope for using declarative programming as a front end to SAT solvers.

We do not illustrate all the different experiments we have tried in solving this problem. We have also experimented with sorting networks for encoding weighted sums (as described in (Eén and Sörensson 2006)), parallel prefix adders (see e.g. (Even 2006)) and many other forms of adders.

Acknowledgments. We would like to thank Daniel Le Berre for interesting discussions and helping us with experiments in this paper. Luis Quesada provided the benchmarks used in (Lesaint et al. 2008) and many useful discussions. Michael Codish acknowledges the support of the Lynn and William Frankel Center for Computer Sciences at Ben-Gurion University. Peter Stuckey is employed by NICTA. NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council. Samir Genaim was supported in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* and IST-231620 *HATS* projects, by the Spanish Ministry of Education (MEC) under the TIN-2005-09207 *MERIT*, TIN-2008-05624 *DOVES* and HI2008-0153 (Acción Integrada) projects, and the Madrid Regional Government under the S-0505/TIC/0407 *PROMESAS* project.

References

- Gregory W. Bond, Eric Cheung, K. Hal Purdy, Pamela Zave, and J. Christopher Ramming. An open architecture for next-generation telecommunication services. *ACM Trans. Internet Techn.*, 4(1):83–123, 2004.
- Michael Codish, Vitaly Lagoon, and Peter J. Stuckey. Telecommunications feature subscription as a partial order constraint problem. In Maria Garcia de la Banda and Enrico Pontelli, editors, *ICLP*, volume 5366 of *Lecture Notes in Computer Science*, pages 749–753. Springer, 2008a. ISBN 978-3-540-89981-5.
- Michael Codish, Vitaly Lagoon, and Peter J. Stuckey. Solving partial order constraints for LPO termination. *Journal on Satisfiability, Boolean Modeling and Computation*, 5:193–215, 2008b. (an earlier version appears in the proceedings of RTA 2006, LNCS 4098).
- Michael Codish, Vitaly Lagoon, and Peter J. Stuckey. Logic programming with satisfiability. *TPLP*, 8(1):121–128, 2008c.
- Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to algorithms*. MIT Press and McGraw-Hill, 1990.
- Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003 (Selected Revised Papers)*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2004.
- Niklas Eén and Niklas Sörensson. Translating pseudo-boolean constraints into sat. *JSAT*, 2(1-4):1–26, 2006.
- Guy Even. On teaching fast adder designs: Revisiting ladner & fischer. In Oded Goldreich, Arnold L. Rosenberg, and Alan L. Selman, editors, *Essays in Memory of Shimon Even*, volume 3895 of *Lecture Notes in Computer Science*, pages 313–347. Springer, 2006. ISBN 3-540-32880-7.
- Carla P. Gomes, Henry Kautz, Ashish Sabharwal, and Bart Selman. *Handbook of Knowledge Representation*, volume 3 of *Foundations of Artificial Intelligence*, chapter Satisfiability Solvers, pages 89–134. Elsevier, 2008. Editors: Frank van Harmelen, Vladimir Lifschitz, and Bruce Porter.
- R. M. Karp. *Reducibility among Combinatorial Problems*, pages 85–103. Plenum Press, 1972. R. E. Miller and J. M. Thatcher (eds.).
- David Lesaint, Deepak Mehta, Barry O’Sullivan, Luis O. Quesada, and Nic Wilson. Solving a telecommunications feature subscription configuration problem. In Peter J. Stuckey, editor, *CP*, volume 5202 of *Lecture Notes in Computer Science*, pages 67–81. Springer, 2008. ISBN 978-3-540-85957-4. (an earlier version appears in the Proceedings of Innovative Applications of Artificial Intelligence, July 2008).
- MiniSAT. MiniSAT. <http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat>. Viewed December 2005.
- Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *DAC*, pages 530–535. ACM, 2001. ISBN 1-58113-297-2.
- Jan Wielemaker. An overview of the SWI-Prolog programming environment. In Fred Mesnard and Alexander Serebenik, editors, *Proceedings of the 13th International Workshop on Logic Programming Environments*, pages 1–16, Heverlee, Belgium, December 2003. Katholieke Universiteit Leuven. CW 371.
- Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *ICCAD '01: Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 279–285, Piscataway, NJ, USA, 2001. IEEE Press. ISBN 0-7803-7249-2.