UNIVERSITY
OF TRENTO

DIPARTIMENTO DI INGEGNERIA E SCIENZA DELL'INFORMAZIONE

38050 Povo – Trento (Italy), Via Sommarive 14
http://www.disi.unitn.it

A SAT-BASED TOOL FOR SOLVING CONFIGURATION PROBLEMS

Stefano Sgorlon

February 2009

Technical Report # DISI-09-013

UNIVERSITÀ DEGLI STUDI DI TRENTO

Facoltà di Scienze Matematiche Fisiche e Naturali

Corso di Laurea Specialistica in Informatica

A SAT-based tool for solving configuration problems

Relatore
Prof. Roberto Sebastiani

Laureando
Stefano Sgorlon

Correlatore
Prof. Zhanshan Li
(*Jilin University, Changchun, China*)

Anno Accademico 2007 - 2008

Contents

Chapter 1

Preface

Configuration problems typically describe situations in which a user has to buy a configurable product selecting its features. The user wants to configure the product in the best way, selecting most of his favourite features and options for the product, but taking into account also the cost and other aspects. The idea is to find the best configuration of the product. Configuration problems are optimization problems in which the "best" solution is found according to some criterion, which in our case is based on the user preferences. They have application in scheduling, configuration, planning, timetabling tasks.

For a simple example of configuration problem we can consider a laptop with some of its features, including the screen size, the screen model, the processor model, the video card and the webcam. The choice of the product features and the budget of the user are expressed by rules, called constraints. A configuration problem describes a product defining variables (the features of the product) and constraints, which are expressions involving the variables. Constraints are necessary to deny some configurations (think for example a very expensive laptop). The constraints are split into two sets, the set of the constraints that must be satisfied (background) and the set of the other remaining constraints (foreground), which can be satisfied or relaxed. In the laptop example the background includes the fundamental features and the constraint regarding the budget, while the foreground includes the optional features.

The work explained here is a tool for solving configuration problems $\langle X, D, B, F\rangle$ where X is a set of variables, D is the set of the variables domains, B is the background and F is the foreground. In particular the tool implements the QuickXplain algorithm (§3.3) for configuration problems on which is defined a binary preference relation $\prec$ among the foreground constraints. This relation expresses the fact that some constraints are more important than other ones, and the satisfaction of the first ones is preferred to the satisfaction of the latter ones.

The QuickXplain algorithm returns the preferred conflict and the preferred relaxation. Given a configuration problem $\langle X,\ D,\ B,\ F,\ \prec \rangle$, the preferred relaxation is a subset of F which is consistent with respect to B, and maximizes the selection of the most important constraints according to $\prec$, while the preferred conflict is a subset of F which is not consistent with respect to B, and minimizes the selection of the least important constraints according to $\prec$.

The problems of finding the preferred relaxation and the preferred conflict for a configuration problem are optimization problems that, in order to compute the "best", here called preferred, consider a lexicographic extension of the preference relation $\prec$. Relaxations and conflicts are dual concepts defined in §3.2.1.

The implemented tool returns also a solution for the configuration problems, namely an assignment variables-values that satisfies the constraints contained in the background and in the preferred relaxation.

QuickXplain is a recursive and dichotomic algorithm that uses a CSP consistency checker to see whether a set of constraints has solutions or not. The idea and the approach here is to use a SAT solver as CSP solver, solving the satisfiability of the formulas encoded from the constraints defined in the configuration problems. MiniSat, an efficient SAT solver, is used for this purpose.

The problems taken into account are written in CP language (§3.4.1), and compose the Configuration Benchmarks Library (CLib) [7]. The tool is implemented in Java and encodes the constrains of every configuration problem into a propositional formula (an instance of the SAT problem) which can solved by a SAT solver.

I started this work in the College of Computer Science and Technology at the Jilin University [2], in Changchun (China). In 2008 I studied there as exchange student under the framework of the EASTWEB project [1].

The chapters of the first part gives a brief introduction on the topics related to this work, mainly SAT and CSP. The second part focuses on the tool, explaining its functionalities and components.

Part I

Background and state of the art

Chapter 2

SAT and SAT solving

The aim of this chapter is to give some ideas and definitions in order to contextualize the implemented tool.

2.1 Basics on propositional logic

A *Boolean variable* is a variable whose value can be $\top$ or $\bot$. These two values are respectively known as *true* and *false*, or, equivalently, 1 and 0. A Boolean variable is called also *Boolean atom*. We use the notation of the capital letters A_i's and B_i's to represent Boolean atoms.

A *propositional formula* (or *Boolean formula*) is a formula composed of Boolean variables, parentheses and the following basics operators:

- $\neg$, called *not*, which represents the negation of a propositional formula
- $\vee$, called *or*, which represents the disjunction of propositional formulas
- $\wedge$, called *and*, which represents the conjunction of propositional formulas
- $\Rightarrow$, called *implies*, which represents the logical implication between two propositional formulas
- $\Leftrightarrow$, called *iff*, which represents the logical equivalence between two propositional formulas

Propositional logic is a formal system in which formulas (representing propositions) can be formed by combining Boolean variables (representing atomic propositions) by means of the above operators. Propositional logic has many applications in electronics, computer hardware and software, and it is the base of digital electronics.

A *literal* is a Boolean variable x or its negation $\neg x$. It is called respectively positive literal or negative literal.

A formula is in *negative normal form* (NNF) if it is written using only the operators $\neg$, $\vee$ and $\wedge$ in such a way that $\neg$ occurs only in front of Boolean variables. Every formula can be translated into an equivalent one in NNF substituting all occurrences of $\Leftrightarrow$ and $\Rightarrow$:

- $\varphi_1 \Leftrightarrow \varphi_2$ is rewritten into $(\varphi_1 \Rightarrow \varphi_2) \wedge (\varphi_2 \Rightarrow \varphi_1)$
- $\varphi_1 \Rightarrow \varphi_2$ is rewritten into $\neg\varphi_1 \vee \varphi_2$

and pushing down negations recursively, applying the following three rules:

- the first De Morgan's law: $\neg(\varphi_1 \wedge \varphi_2)$ is rewritten into $\neg\varphi_1 \vee \neg\varphi_2$
- the second De Morgan's law: $\neg(\varphi_1 \vee \varphi_2)$ is rewritten into $\neg\varphi_1 \wedge \neg\varphi_2$
- the double negative law: $\neg\neg\varphi_1$ is rewritten into φ_1

A propositional formula can be represented as a tree or as a directed acyclic graph (DAG).

A *total (partial) truth assignment* is a function that maps all the (some) variables of a propositional formula into $\{\top, \bot\}$.

A *clause* is a disjunction of literals. If l_1, l_2, $\ldots$, l_n are literals, a simple example of a clause is the propositional formula $l_1 \vee l_2 \vee \cdots \vee l_n$. A clause composed of a single literal is called unit clause.

Given a propositional formula φ, the *polarity* of a sub-formula ψ contained in φ is *positive (negative)* if ψ occurs in φ under the scope of an even (odd) number of negations. A sub-formula is called *positive (negative)* if its polarity is positive (negative).

Given a propositional formula φ, and two arbitrary sub-formulas φ_1 and φ_2 contained in φ, we have that:

- φ occurs positively in φ

- if $\neg\varphi_1$ occurs positively (negatively) in φ, then φ_1 occurs negatively (positively) in φ
- if $\varphi_1 \wedge \varphi_2$ or $\varphi_1 \vee \varphi_2$ occur positively (negatively) in φ, then φ_1 and φ_2 occur positively (negatively) in φ
- if $\varphi_1 \Rightarrow \varphi_2$ occurs positively (negatively) in φ, then φ_1 occurs negatively (positively) in φ and φ_2 occurs positively (negatively) in φ
- if $\varphi_1 \Leftrightarrow \varphi_2$ occurs in φ, then φ_1 and φ_2 occur both positively and negatively in φ

2.2 SAT

Satisfiability is the problem of determining if the variables of a given propositional formula can be assigned in a way that makes the formula evaluated to *true*. Equally important is to determine whether no such assignments exist, which would imply that the function expressed by the formula is identically *false* for all possible variable assignments. In this latter case, we would say that the function is unsatisfiable; otherwise it is satisfiable. The satisfiability problem is also known as SAT.

More formally, a propositional formula φ is *satisfiable* if and only if there exists a truth assignment μ that makes φ evaluated to $\top$. If such assignment exists, μ is called model for the formula φ and we write $\mu \models \varphi$. Dually, φ is *unsatisfiable* if and only if there are no models for φ, or, equivalently, if and only if all truth assignments of φ make the formula evaluated to $\bot$.

The formula φ is *valid* if and only if all the possible truth assignments make φ evaluated to *true*.

Two propositional formulas φ and ϑ are *equivalent* if and only if, for every truth assignment μ, we have that $\mu \models \varphi \Leftrightarrow \mu \models \vartheta$.

In complexity theory, SAT is a decision problem, whose instance is a propositional formula written using only $\wedge$, $\vee$ $\neg$, variables, and parentheses. The question is: given a propositional formula, is there an assignment of *true* and *false* values to the variables that make the entire expression *true*? The satisfiability problem is an *NP*-complete problem [17]. Hence, there are no known polynomial algorithms for solving SAT. This problem is of central importance in various areas of computer science, including theoretical computer science, algorithmics, artificial intelligence, hardware design, electronic design automation, and verification.

Determining whether a propositional formula in CNF is satisfiable is still NP-complete.

2.3 Conjunctive normal form

A propositional formula is in *conjunctive normal form* (CNF) if it is a conjunction of clauses. Conjunctions of literals and disjunctions of literals are in CNF, as they can be seen as conjunctions of one-literal clauses and disjunctions of a single clause, respectively.

2.3.1 Classical CNF conversion

Every propositional formula can be converted into a CNF equivalent formula. The classical conversion is based on the NNF rules and on the following rules:

- the first distributive law: $\varphi_1 \vee (\varphi_2 \wedge \varphi_3)$ is rewritten into $(\varphi_1 \vee \varphi_2) \wedge (\varphi_1 \vee \varphi_3)$
- the second distributive law: $\varphi_1 \wedge (\varphi_2 \vee \varphi_3)$ is rewritten into $(\varphi_1 \wedge \varphi_2) \vee (\varphi_1 \wedge \varphi_3)$

This conversion preserves the validity of formulas, namely φ is valid if and only if its correspondent CNF formula, computed using the classical conversion, is valid. The classical CNF transformation is rarely used because it is exponential in the worth case. In the next session it is explained a faster CNF conversion, linear in the worst case: the labeling CNF conversion.

2.3.2 Labeling CNF conversion

The labeling CNF conversion is a faster CNF conversion that preserves the satisfiability of formulas, namely φ is satisfiable if and only if its correspondent CNF formula, computed using the labeling conversion, is satisfiable. It can be applied for every propositional formula.

This conversion introduces a label, which is a new Boolean variable, for each nontrivial sub-formula. Given a formula φ, and a sub-formula ψ contained in φ, the labeling CNF conversion draws on this fact:

- φ is encoded into $\varphi_{[\psi|H]} \wedge CNF^*(\psi \Leftrightarrow H)$

where $\varphi_{[\psi|H]}$ represents the formula φ in which the label H replaces all the instances of ψ in φ, and $CNF^*(\vartheta)$ is the formula ϑ converted in CNF applying the classical rules (see §2.3).

In particular, the sub-formula ψ contained in φ can be a disjunction, a conjunction, a logical implication, or a logical equivalence. Hence, computing the classical CNF conversion of ψ, for every literal l_1, l_2, $\dots$, l_n, we have that:

- $CNF^*((l_1 \vee l_2 \vee \cdots \vee l_n) \Leftrightarrow H)$ is encoded into $(\neg l_1 \vee H) \wedge (\neg l_2 \vee H) \wedge \cdots \wedge (\neg l_n \vee H) \wedge (l_1 \vee l_2 \vee \cdots \vee l_n \vee \neg H)$
- $CNF^*((l_1 \wedge l_2 \wedge \cdots \wedge l_n) \Leftrightarrow H)$ is encoded into $(l_1 \vee \neg H) \wedge (l_2 \vee \neg H) \wedge \cdots \wedge (l_n \vee \neg H) \wedge (\neg l_1 \vee \neg l_2 \vee \cdots \vee \neg l_n \vee H)$
- $CNF^*((l_1 \Rightarrow l_2) \Leftrightarrow H)$ is encoded into $(l_1 \vee H) \wedge (\neg l_2 \vee H) \wedge (\neg l_1 \vee l_2 \vee \neg H)$
- $CNF^*((l_1 \Leftrightarrow l_2) \Leftrightarrow H)$ is encoded into $(\neg l_1 \vee l_2 \vee \neg H) \wedge (l_1 \vee \neg l_2 \vee \neg H) \wedge (l_1 \vee l_2 \vee H) \wedge (\neg l_1 \vee \neg l_2 \vee H)$

The algorithm for computing the labeling CNF conversion is recursive. For each formula ϑ it introduces a label H (a new Boolean variable) that replaces ϑ, and it adds the condition $\vartheta \Leftrightarrow H$ (ϑ is *true* if and only if H is *true*). This is done recursively for all the nested sub-formulas, starting from the whole formula φ, and ending when the formula to analyze is atomic or it can be trivially converted into an equivalent CNF formula using the classical conversion.

Let's see now how the algorithm works for a simple example.

Example 2.3.1. *Let's take the formula $\neg x_1 \vee (x_2 \wedge x_3 \wedge x_4 \wedge (x_5 \Leftrightarrow x_6))$, where x_1, x_2, x_3, x_4, x_5, x_6 are Boolean variables. In order to label the sub-formulas, the algorithm introduces chronologically three new variables: b_1, b_2 and b_3.*

In particular:

- *b_1 is the label for the whole formula $\neg x_1 \vee (x_2 \wedge x_3 \wedge x_4 \wedge (x_5 \Leftrightarrow x_6))$*
- *b_2 is the label for the sub-formula $x_2 \wedge x_3 \wedge x_4 \wedge (x_5 \Leftrightarrow x_6)$*
- *b_3 is the label for the sub-formula $x_5 \Leftrightarrow x_6$*

At the beginning the label b_1 is introduced. Then, in the next steps, b_2 and b_3 are used to label the other two internal sub-formulas. The resulting CNF formula given by this conversion is $b_1 \wedge CNF^*((\neg x_1 \vee b_2) \Leftrightarrow b_1) \wedge CNF^*((x_2 \wedge x_3 \wedge x_4 \wedge b_3) \Leftrightarrow b_2) \wedge CNF^*((x_5 \Leftrightarrow x_6) \Leftrightarrow b_3)$, where:

- *$CNF^*((\neg x_1 \vee b_2) \Leftrightarrow b_1)$ is rewritten into $(x_1 \vee b_1) \wedge (\neg b_2 \vee b_1) \wedge (\neg x_1 \vee b_2 \vee \neg b_1)$*
- *$CNF^*((x_2 \wedge x_3 \wedge x_4 \wedge b_3) \Leftrightarrow b_2)$ is rewritten into $(x_2 \vee \neg b_2) \wedge (x_3 \vee \neg b_2) \wedge (x_4 \vee \neg b_2) \wedge (b_3 \vee \neg b_2) \wedge (\neg x_2 \vee \neg x_3 \vee \neg x_4 \vee \neg b_3 \vee b_2)$*
- *$CNF^*((x_5 \Leftrightarrow x_6) \Leftrightarrow b_3)$ is rewritten into $(\neg x_5 \vee x_6 \vee b_3) \wedge (x_5 \vee \neg x_6 \vee b_3) \wedge (x_5 \vee x_6 \vee \neg b_3) \wedge (\neg x_5 \vee \neg x_6 \vee \neg b_3)$*

The labeling CNF conversion can be improved if we take into account the polarity of the sub-formulas. In fact, we have that:

- if ψ is positive, then φ is rewritten into $\varphi_{[\psi|H]} \wedge CNF^*(H \Rightarrow \psi)$
- if ψ is negative, then φ is rewritten into $\varphi_{[\psi|H]} \wedge CNF^*(\psi \Rightarrow H)$

This improved version returns a CNF formula with a smaller number of literals. In fact:

- $CNF^*((l_1 \vee l_2 \vee \cdots \vee l_n) \Rightarrow H)$ is rewritten into $(\neg l_1 \vee H) \wedge (\neg l_2 \vee H) \wedge \cdots \wedge (\neg l_n \vee H)$
- $CNF^*(H \Rightarrow (l_1 \vee l_2 \vee \cdots \vee l_n))$ is rewritten into $l_1 \vee l_2 \vee \cdots \vee l_n \vee \neg H$
- $CNF^*((l_1 \wedge l_2 \wedge \cdots \wedge l_n) \Rightarrow H)$ is rewritten into $\neg l_1 \vee \neg l_2 \vee \cdots \vee \neg l_n \vee H$
- $CNF^*(H \Rightarrow (l_1 \wedge l_2 \wedge \cdots \wedge l_n))$ is rewritten into $(l_1 \vee \neg H) \wedge (l_2 \vee \neg H) \wedge \cdots \wedge (l_n \vee \neg H)$
- $CNF^*((l_1 \Rightarrow l_2) \Rightarrow H)$ is rewritten into $(l_1 \vee H) \wedge (\neg l_2 \vee H)$
- $CNF^*(H \Rightarrow (l_1 \Rightarrow l_2))$ is rewritten into $\neg l_1 \vee l_2 \vee \neg H$
- $CNF^*((l_1 \Leftrightarrow l_2) \Rightarrow H)$ is rewritten into $(l_1 \vee l_2 \vee H) \wedge (\neg l_1 \vee \neg l_2 \vee H)$
- $CNF^*(H \Rightarrow (l_1 \Leftrightarrow l_2))$ is rewritten into $(\neg l_1 \vee l_2 \vee \neg H) \wedge (l_1 \vee \neg l_2 \vee \neg H)$

2.3.3 DIMACS format

DIMACS format is widely accepted as the standard format for propositional formulas in CNF.

An input file in DIMACS format starts with comments (each line begins with the lower case letter 'c'). The number of variables and the number of clauses are defined by the line

```
p cnf variables clauses
```

Each of the next lines specifies a clause: a positive literal is denoted by the corresponding number, and a negative literal is denoted by the corresponding negative number. The last number in a line should be zero.

For example, to express in DIMACS format the CNF formula $(x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_1 \vee \neg x_3)$, we write a text file containing these lines:

```
c A simple DIMACS CNF file
p cnf 3 2
1 -2 0
2 -1 -3 0
```

2.4 The DPLL algorithm

The DPLL (Davis-Putnam-Logemann-Loveland) algorithm is a backtracking-based algorithm for solving the SAT problem. DPLL is a highly efficient procedure that forms the basis for most efficient SAT solvers.

The basic backtracking algorithm runs by choosing a literal, assigning a truth value to it, simplifying the formula and then recursively checking if the simplified formula is satisfiable. If this is the case, the original formula is satisfiable, otherwise, the same recursive check is done assuming the opposite truth value. This is known as the splitting rule, as it splits the problem into two simpler sub-problems. The simplification step essentially removes all clauses which become true under the assignment from the formula, and all literals that become false from the remaining clauses.

The DPLL algorithm enhances over the backtracking algorithm using at each step two procedures: the *unit propagation* and the *pure literal elimination.* Unit propagation is based on the fact that if a clause is a unit clause, then it can only be satisfied by assigning the necessary value to make its literal true.

Thus, no choice is necessary on this literal. In practice, this often leads to deterministic cascades of unit propagations, thus avoiding a large part of the naive search space. Pure literal elimination is based on the fact that pure literals [1] can always be assigned in a way that makes all clauses containing them true. Thus, these clauses do not constrain the search anymore and can be deleted. While this optimization is part of the original DPLL algorithm, most current implementations omit it, because the effect for efficient implementations now is negligible or, due to the overhead for detecting purity, even negative.

The unsatisfiability of a given partial assignment is detected if one clause becomes empty, namely if all its variables have been assigned in a way that makes the corresponding literals false. The satisfiability of the formula is detected either when all variables are assigned without generating the empty clause, or, in modern implementations, if all clauses are satisfied. The unsatisfiability of the complete formula can only be detected after exhaustive search.

The DPLL algorithm can be summarized in the following pseudocode, where φ is the CNF formula.

Algorithm 2.4.1: DPLL(φ)

```
if $\varphi$ is a consistent set of literals
  then return ($true$)
  else
    if $\varphi$ contains an empty clause
      then return ($false$)
      else
        while $\varphi$ has unit clauses
          do
            $l :=$ next unit clause in $\varphi$
            $\varphi :=$ unit-propagate$(l, \varphi)$
        while $\varphi$ has pure literals
          do
            $l :=$ next pure literal in $\varphi$
            $\varphi :=$ pure-literal-assign$(l, \varphi)$
        $l :=$ choose-literal$(\varphi)$
        if DPLL$(\varphi \wedge l)$
          then return ($true$)
          else return (DPLL$(\varphi \wedge \neg l)$)
```

In this pseudocode, *unit-propagate(l, φ)* and *pure-literal-assign(l, φ)* are functions that return the result of applying unit propagation and the pure literal

[1]Given a propositional formula φ, a Boolean variable is called *pure* if it occurs with only one polarity in φ.

procedure, respectively, to the literal l and the formula φ. In other words, they replace every occurrence of l with *true* and every occurrence of $\neg l$ with *false* in the formula φ, and simplify the resulting formula. The DPLL function only returns whether the final assignment satisfies the formula or not. In a real implementation, the partial satisfying assignment typically is also returned on success. This can be derived from the consistent set of literals of the first if-statement of the function.

The DPLL algorithm depends on the choice of the branching literal, which is the literal considered in the backtracking step. For this reason DPLL is a family of algorithms, one for each possible way of choosing the branching literal. Efficiency is strongly affected by the choice of the branching literal: there exist instances for which the running time is constant or exponential depending on the choice of the branching literals.

2.4.1 Modern conflict-driven DPLL

This section is taken from [22, 23].

A SAT solver is a procedure which decides whether an input Boolean formula φ is satisfiable, and returns a satisfying assignment if this is the case.

Most state-of-the-art SAT procedures are evolutions of the DPLL procedure [30, 29]. Unlike with "classic" representation of DPLL [30, 29], modern conflict-driven DPLL implementation are non-recursive, and are based on very efficient data structures to handle Boolean formulas and assignments. They benefit of sophisticated search techniques, smart decision heuristics, highly-engineered data structures and cute implementation tricks, and smart preprocessing techniques [46].

Modern DPLL engines can be partitioned into two main families: conflict-driven DPLL [43], in which the search is driven by the analysis of the conflicts at every failed branch, and look-ahead DPLL [37], in which the search is driven by a look-ahead procedure evaluating the reduction effect of the selection of each variable in a group. Hereafter we restrict our discussion to the conflict-driven DPLL schema, and in the next sections we often omit the adjective "conflict-driven" when referring to DPLL.

A high-level schema of a modern conflict-driven DPLL engine, adapted from [46], is reported in Figure 2.1. The Boolean formula φ is in CNF; the assignment μ is initially empty, and it is updated in a stack-based manner. The function *preprocess*(φ, μ) simplifies φ into a simpler and equi-satisfiable

```
SatValue DPLL (Bool_formula $\varphi$, assignment & $\mu$) {
    if (preprocess($\varphi, \mu$)==Conflict);
        return Unsat;
    while (1) {
        decide_next_branch($\varphi, \mu$);
        while (1) {
            status = deduce($\varphi, \mu$);
            if (status == Sat)
                return Sat;
            else if (status == Conflict) {
                blevel = analyze_conflict($\varphi, \mu$);
                if (blevel == 0)
                    return Unsat;
                else backtrack(blevel,$\varphi, \mu$);
            }
            else break;
} } }
```

Figure 2.1: Schema of a modern conflict-driven DPLL engine.

formula, and updates μ if it is the case [2] . If the resulting formula is unsatisfiable, then DPLL returns Unsat.

In the main loop, *decide_next_branch(φ, μ)* chooses an unassigned literal l from φ according to some heuristic criterion, and adds it to μ. This operation is called decision, l is called *decision literal* and the number of decision literals in μ after this operation is called the *decision level* of l.

In the inner loop, *deduce(φ, μ)* iteratively deduces literals l deriving from the current assignments and updates φ and μ accordingly; this step is repeated until either μ satisfies φ, or μ falsifies φ, or no more literals can be deduced, returning Sat, Conflict and Unknown respectively. The iterative application of Boolean deduction steps in *deduce* is also called *Boolean constraint propagation* (BCP).

In the first case, DPLL returns Sat. In the second case, *analyze_conflict(φ, μ)* detects the subset η of μ which caused the conflict (*conflict set*) and the decision level *blevel* to backtrack. If *blevel*==0, then a conflict exists even without branching, so that DPLL returns Unsat. Otherwise, *backtrack(blevel,φ, μ)* adds $\neg\eta$ to φ (learning) and backtracks up to *blevel* (backjumping), updating

[2] More precisely, if φ, μ, φ', μ' are the formula and the assignment before and after preprocessing respectively, then $\varphi' \wedge \mu'$ is equisatisfiable to $\varphi \wedge \mu$.

φ and μ accordingly. In the third case, DPLL exits the inner loop, looking for the next decision.

We look at these steps with some more detail.

The function *preprocess* implements simplification techniques like, e.g., detecting and inlining Boolean equivalences among literals, applying resolutions steps to selected pairs of clauses, detecting and dropping subsumed clauses (see, e.g., [25, 27, 31]). It may also apply BCP if this is the case.

The function *decide_next_branch* implements the key non-deterministic step in DPLL for which many heuristic criteria have been conceived. Old-style heuristics like MOMS and Jeroslow-Wang [35] used to select a new literal at each branching point, picking the literal occurring most often in the minimal-size clauses (see, e.g., [34]). The heuristic implemented in SATZ [36] selects a candidate set of literals, performs BCP, chooses the one leading to the smallest clause set; this maximizes the effects of BCP, but introduces big overheads. When formulas derive from the encoding of some specific problem, it is sometimes useful to allow the encoder to provide to the DPLL solver a list of "privileged" variables on which to branch first (e.g., action variables in SAT-based planning [32], primary inputs in bounded model checking [44]). Modern conflict-driven DPLL solvers adopt evolutions of the VSIDS heuristic [39, 24, 13], in which decision literals are selected according to a score which is updated only at the end of a branch, and which privileges variables occurring in recently-learned clauses; this makes *decide_next_branch* state-independent (and thus much faster, because there is no need to recomputing the scores at each decision) and allows it to take into account search history, which makes search more effective and robust.

The function *deduce* is mostly based on the iterative application of unit-propagation. Highly-engineered data structures and cute implementation tricks (like the two-watched-literal scheme [39]) allow for extremely efficient implementations. Other forms of deductions (and formula simplification) are, e.g., pure literal rule (now obsolete), on-line equivalence reasoning [38], and variable and clause elimination [31].

It is important to notice that most modern conflict-driven DPLL solvers do not return Sat when all clauses are satisfied, but only when all variables are assigned truth values [3] . As a consequence, modern conflict-driven SAT solvers typically return total truth assignments, even though the formulas are satisfied by partial ones.

[3] This is mostly due to the fact that the two-watched-literal scheme [39] does not allow for an easy check of clause satisfaction (e.g., if a non-watched literal l in the clause $C \vee l$ is true, then the clause it satisfied but DPLL is not informed of this fact).

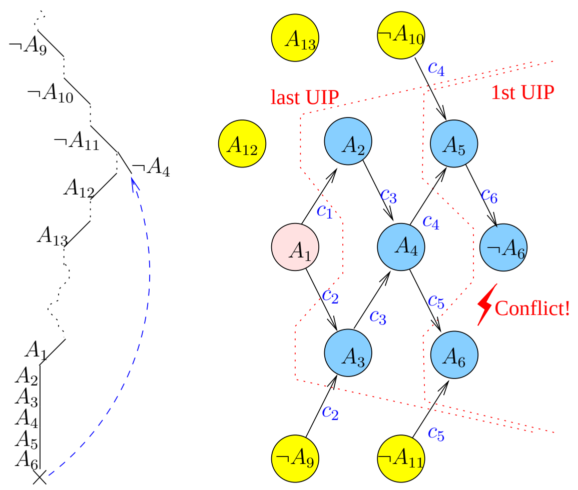
The functions *analyze_conflict* and *backtrack* work as follows [43, 26, 47, 46]. Each literal is tagged with its decision level, that is, the literal corresponding to the nth decision and the literals derived by unit-propagation after that decision are labeled with n; each non-decision literal l in μ is tagged by a link to the clause C_l causing its unit-propagation (called the antecedent clause of l). When a clause C is falsified by the current assignment, in which case we say that a conflict occurs and C is the conflicting clause, a conflict clause C' is computed from C such that C' contains only one literal l_u which has been assigned at the last decision level. C' is computed starting from $C' = C$ by iteratively resolving C' with the antecedent clause C_l of some literal l in C' (typically the last-assigned literal in C', see [46]), until some stop criterion is met. E.g., with the first UIP strategy it is always picked the last-assigned literal in C', and the process stops as soon as C' contains only one literal l_u assigned at the last decision level; with the last UIP strategy, l_u must be the last decision literal.

Graphically, building a conflict set/clause corresponds to (implicitly) building and analyzing the *implication graph* corresponding to the current assignment. An implication graph is a DAG such that each node represents a variable assignment (literal), the node of a decision literal has no incoming edges, all edges incoming into a non-decision-literal node l are labeled with the antecedent clause C_l, such that $l_1 \xmapsto{C_l} l, ..., l_n \xmapsto{C_l} l$ if and only if $C_l = \neg l_1 \vee ... \vee \neg l_n \vee l$. When both l and $\neg l$ occur in the implication graph we have a conflict; given a partition of the graph with all decision literals on one side and the conflict on the other, the set of the source nodes of all arcs intersecting the borderline of the partition represents a conflict set. A node l_u in an implication graph is a *unique implication point* (UIP) for the last decision level if and only if any path from the last decision node to both the conflict nodes passes through l_u [4] ; the most recent decision node is a UIP (the *last* UIP); the most-recently-assigned UIP is called the *first* UIP. E.g., the last (first) UIP strategy corresponds to using as conflict set a partition corresponding to the last (first) UIP.

After *analyze_conflict* has computed the conflict clause C' and added it to the formula, *backtrack* pops all assigned literals out of μ up to a decision level *blevel* deriving from C', which is computed by *analyze_conflict* according to different strategies. In modern conflict-driven implementations, DPLL backtracks to the highest point in the stack where the literal l_u in the learned clause C' is not assigned, and unit-propagates l_u. We refer the reader to [47] for an overview on backjumping and learning strategies.

[4] An UIP is also called an *articulation point* in graph theory (see, e.g., [28]).

$c_1 : \{\neg A_1 \vee A_2\}$ $\checkmark$
$c_2 : \{\neg A_1 \vee A_3 \vee A_9\}$ $\checkmark$
$c_3 : \{\neg A_2 \vee \neg A_3 \vee A_4\}$ $\checkmark$
$c_4 : \{\neg A_4 \vee A_5 \vee A_{10}\}$ $\checkmark$
$c_5 : \{\neg A_4 \vee A_6 \vee A_{11}\}$ $\checkmark$
$c_6 : \{\neg A_5 \vee \neg A_6\}$ $\times$
$c_7 : \{A_1 \vee A_7 \vee \neg A_{12}\}$ $\checkmark$
$c_8 : \{A_1 \vee A_8\}$ $\checkmark$
$c_9 : \{\neg A_7 \vee \neg A_8 \vee \neg A_{13}\}$
...



$\Longrightarrow$Conflict set: $\{\neg A_{10}, \neg A_{11}, A_4\}$, learn $c_{10} := A_{10} \vee A_{11} \vee \neg A_4$

Figure 2.2: Example of learning and backjumping based on the first UIP stategy.

Example 2.4.1. *Consider a Boolean formula containing the clauses $c_1 \dots c_9$ in Figure 2.2, and assume at some point $\mu := \{\dots, \neg A_9, \dots, \neg A_{10}, \dots, \neg A_{11}, \dots, A_{12}, \dots, A_{13}, \dots, A_1\}$ [5] . After applying BCP on $c_1 \dots c_8$ a conflict on c_6 occurs. Starting from the conflicting clause c_6, the conflict clause/set is computed by iteratively resolving the current clause C' each time with the antecedent clause of the last-assigned literal l in C', until it contains only one literal assigned at the current decision level (first UIP):*

$$\dfrac{\overbrace{\neg A_4 \vee A_5 \vee A_{10}}^{c_4} \qquad \dfrac{\overbrace{\neg A_4 \vee A_6 \vee A_{11}}^{c_5} \qquad \overbrace{\neg A_5 \vee \neg A_6}^{conflicting\ clause}}{\neg A_4 \vee \neg A_5 \vee A_{11}}\ (A_6)}{\underbrace{\neg A_4}_{1st\ UIP} \vee A_{10} \vee A_{11}}\ (A_5)$$

This corresponds to the first UIP cut of the implication graph in Figure 2.2. Then DPLL learns the conflict clause $c_{10} := A_{10} \vee A_{11} \vee \neg A_4$, and backtracks up to below $\neg A_{11}$, it unit-propagates $\neg A_4$ on c_{10}, and proceeds.

Learning must be used with some care, because it may cause an explosion in the size of φ. In order to avoid this problem, modern conflict-driven DPLL tools implement techniques for discharging learned clauses when necessary

[5] Here and in other examples "..." mean that there may be possibly many other literals in the assignment, which play no direct role in the discourse.

[43, 26]. Moreover, in order to avoid getting stuck into hard portions of the search space, most DPLL tools restart the search from scratch in a controlled manner [33]; the clauses which have been learned avoid exploring the same search tree again.

2.4.2 The Abstract-DPLL logical framework

This section is taken from [22].

[45, 41, 40, 42] proposed an abstract rule-based formulation of DPLL (Abstract DPLL). In this framework, DPLL is modeled as a transition system. A state is either $fail$ or a pair $\langle\mu \mid \varphi\rangle$, φ being a CNF Boolean formula and μ being a set of annotated literals, representing the current truth assignment. All DPLL steps are seen as transitions in the form $\langle\mu \mid \varphi\rangle \Rightarrow \langle\mu' \mid \varphi'\rangle$, and are applications of the conditioned transition rules described in Figure 2.3 [6] . The first five rules represent respectively the unit-propagation step of *deduce*, the literal selection in *decide_next_branch*, the failure step of row 12-13 in Figure 2.1, the backjumping and learning mechanisms of *analyze_conflict* and *backtrack*. The last two rules represent the discharging and restart mechanisms described, e.g., in [26] and [33].

The only non-obvious rule is Backjump, which deserves some more explanation: if a branch $\mu \cup \{l\} \cup \mu'$ falsifies one clause C (the conflicting clause), and a conflict clause $C' \vee l'$ [7] can be computed from C such that $(C' \vee l')$ is entailed by $\varphi \wedge C$, $\neg C' \subseteq \mu$, $l' \not\in \mu$, and l' or $\neg l'$ occur in φ or in $\mu \cup \{l\} \cup \mu'$, then it is possible to backjump up to μ, and hence unit-propagate l' on the conflict clause $(C' \vee l')$.

Example 2.4.2. *Consider the problem in Example 2.4.1 and Figure 2.2.*

[6]The formalization of the rules in [41, 40, 42] changes slightly from paper to paper. Here we report the most-recent one from [42].

[7]Also called the backjump clause in [42].

$$if \begin{cases} \mu \models \neg C \\ l \; is \; undefined \; in \; \mu \end{cases}$$
Unit-Propagate: $\langle \mu \mid \varphi, C \vee l \rangle \;\; \Rightarrow \langle \mu, l \mid \varphi, C \vee l \rangle$

$$if \begin{cases} l \; or \; \neg l \; occurs \; in \; \varphi \\ l \; is \; undefined \; in \; \mu \end{cases}$$
Decide: $\langle \mu \mid \varphi \rangle \;\; \Rightarrow \langle \mu, l \mid \varphi \rangle$

$$if \begin{cases} \mu \models \neg C \\ \mu \; contains \; no \; decision \; literals \end{cases}$$
Fail: $\langle \mu \mid \varphi, C \rangle \;\; \Rightarrow fail$

$$if \begin{cases} \mu, l, \mu' \models \neg C \\ there \; is \; some \; clause \; C' \vee \; l' \; s.t. : \\ \quad \varphi, C \models C' \vee l' \; and \; \mu \models \neg C' \\ \quad l' \; is \; undefined \; in \; \mu \\ \quad l' \; or \; \neg l' \; occurs \; in \; \varphi \; or \\ \quad\; in \; \mu \cup \{l\} \cup \mu' \end{cases}$$
Backjump: $\langle \mu, l, \mu' \mid \varphi, C \rangle \;\; \Rightarrow \langle \mu, l' \mid \varphi, C \rangle$

$$if \begin{cases} all \; atoms \; in \; C \; occur \; in \; \varphi \; or \; in \; \mu \\ \varphi \models C \end{cases}$$
Learn: $\langle \mu \mid \varphi \rangle \;\; \Rightarrow \langle \mu \mid \varphi, C \rangle$

$if \; \{ \; \varphi \models C$
Discharge: $\langle \mu \mid \varphi, C \rangle \;\; \Rightarrow \langle \mu \mid \varphi \rangle$

Restart: $\langle \mu \mid \varphi \rangle \;\; \Rightarrow \langle \emptyset \mid \varphi \rangle$

Figure 2.3: The Abstract-DPLL logical framework from [42]. In the Backjump rule, C and $C' \vee l'$ represent the conflicting and the conflict clause respectively.

The execution can be represented in Abstract-DPLL as follows:

...

$\langle .., \neg A_9, .., \neg A_{10}, .., \neg A_{11}, .., A_{12}, .., A_{13}, .. \quad | c_1, ..., c_9 \rangle$
$\Rightarrow$ (*Decide* A_1)

$\langle .., \neg A_9, .., \neg A_{10}, .., \neg A_{11}, .., A_{12}, .., A_{13}, .., A_1 \quad | c_1, ..., c_9 \rangle$
$\Rightarrow$ (*UnitP.* A_2)

$\langle .., \neg A_9, .., \neg A_{10}, .., \neg A_{11}, .., A_{12}, .., A_{13}, .., A_1, A_2 \quad | c_1, ..., c_9 \rangle$
$\Rightarrow$ (*UnitP.* A_3)

...

$\langle .., \neg A_9, .., \neg A_{10}, .., \neg A_{11}, .., A_{12}, .., A_{13}, .., A_1, A_2, A_3, A_4, A_5, A_6 | c_1, ..., c_9 \rangle$
$\Rightarrow$ (*Learn* c_{10})

$\langle .., \neg A_9, .., \neg A_{10}, .., \neg A_{11}, .., A_{12}, .., A_{13}, .., A_1, A_2, A_3, A_4, A_5, A_6 | c_1, ..., c_9, c_{10} \rangle$
$\Rightarrow$ (*Backjump*)

$\langle .., \neg A_9, .., \neg A_{10}, .., \neg A_{11}, \neg A_1 \quad | c_1, ..., c_9, c_{10} \rangle$
$\Rightarrow$ (...)

...

$c_1, ..., c_{10}$ *being the clauses in Figure 2.2.*

If a finite sequence $\langle \emptyset \mid \varphi \rangle \Rightarrow \langle \mu_1 \mid \varphi_1 \rangle \Rightarrow \ldots \Rightarrow fail$ is found, then the formula is unsatisfiable; if a finite sequence $\langle \emptyset \mid \varphi \rangle \Rightarrow \ldots \Rightarrow \langle \mu_n \mid \varphi_n \rangle$ is found so that no rule can be further applied, then the formula is satisfiable. Different strategies in applying the rules correspond to different variants of the algorithm. [41, 42] provide a group of results about termination, correctness and completeness of various configurations. Importantly, notice only the second, third and fourth rules are strictly necessary for correctness and completeness [41]. We refer the reader to [41, 40, 42] for further details.

2.5 The MiniSat solver

MiniSat is at the moment one of the most efficient SAT solvers. MiniSat was recently awarded in the competitive events for SAT solvers, SAT Competition [11] and SAT-Race 2006 [12]. It employs conflict-driven learning and uses a two-watched-literal scheme [39] for efficient BCP. It is an open-source SAT solver. Easy to modify, it can be integrated as a backend to another tool that needs a SAT solver.

MiniSat solves satisfiability problems that are in DIMACS conjunctive normal form (see §2.3.3).

In order to run MiniSat we need to launch the MiniSat executable file from a shell passing as argument the path of the file that stores the DIMACS CNF formula we want to solve the satisfiability. MiniSat checks the formula and returns Sat if the formula has a model, otherwise it returns Unsat. It is possible also to obtain the output in a file, passing to MiniSat a second argument, the path of the output file. MiniSat will print in the file Sat or Unsat, depending on the satisfiability of the formula. If the formula is satisfiable, MiniSat will return also the model that it has found for the formula.

Let's see now how MiniSat works internally [13]. First MiniSat creates an instance of the solver. Then it adds many variables as indicated in the first valid (not commented) row of the input file. The variables indexes are stored in a queue. Then it adds one clause at a time in a list. After the addition of each clause, MiniSat checks if there are some trivial conflicts between the current clause and the previous ones (for example it is able to detect the conflict between the simple clauses x_1 and $\neg x_1$). If a conflict is detected, MiniSat exits and returns Unsat.

Otherwise, when all the clauses are added to the solver, MiniSat makes some simplifications on the original formula. Also here MiniSat can sometimes detect conflicts and exit returning Unsat.

At the end it solves the satisfiability problem of the simplified propositional formula, and it returns Sat or Unsat.

Every time MiniSat resets its data structures for a new SAT instance to solve. If we have to solve a set of related SAT instances $\{\phi_1, \phi_2, \ldots, \phi_n\}$, which can share variables and clauses, we may use MiniSat incrementally, achieving a big performance gain.

A useful feature of MiniSAT is the so called *incremental* SAT solving. The idea behind this is that one uses the solver to find a solution for a general problem and then uses the result from this computation to solve similar, more specific problems in a very short time. Instead of adding the additional problem clauses directly, they are passed to the solver as a parameter of additional clauses, called assumptions. The advantage of this approach is that the solver can easily distinguish the general problem clauses from the special problem clauses, so if the problem is unsatisfiable the solver can return to the initial, general problem state by discarding all changes that were made based on the assumptions. In order to perform incremental SAT solving in MiniSat we need to use the source code of MiniSat. We have to write a program in C or C++ that uses the application programming interface (API) of MiniSat.

Chapter 3

CSPs and configuration problems

3.1 Basics on constraint programming

Constraint programming is a powerful paradigm for solving combinatorial search problems that draws on a wide range of techniques from artificial intelligence, computer science, databases, programming languages, and operations research. Constraint programming is currently applied with success to many domains, such as scheduling, planning, vehicle routing, configuration, networks, and bioinformatics.

Constraint programming is the use of constraints as a programming language to encode and solve problems. This is often done by embedding constraints into a programming language, which is called the host language.

Constraints differ from the common primitives of imperative programming languages in that they do not specify a step or a sequence of steps to execute, but rather the properties of a solution to be found. This makes the constraint programming a form of declarative programming. Constraint programming began with constraint logic programming, which embeds constraints into a logic program.

The constraints used in constraint programming are of various kinds, and are typically over some specific domains. Some popular domains for constraint programming are:

- Boolean domains
- integer domains

- linear domains
- finite domains
- mixed domains

Finite domains is one of the most successful domains of constraint programming. Finite domain solvers are useful for solving constraint satisfaction problems.

3.1.1 Constraint satisfaction

Some parts here follow [15]. In artificial intelligence and operations research, constraint satisfaction is the process of finding a solution to a set of constraints that impose some conditions on a certain number of variables. A solution is therefore a set of assignments of values to each variable that satisfies all the constraints.

The techniques used in constraint satisfaction depend on the kind of constraints. There are many possible kind of constraints. Often used are constraints on a finite domain.

Constraint satisfaction problems on finite domains are typically solved using search algorithms. The most used techniques are variants of backtracking, constraint propagation, and local search. The latter techniques are used on problems with nonlinear constraints.

Variable elimination and the simplex algorithm are used for solving linear and polynomial equations and inequalities, and problems containing variables with infinite domain.

Many problems in computer science and mathematics can be formulated as constraint satisfaction problems. Moreover, many real-life problems can be represented as constraint satisfaction problems.

A *constraint satisfaction problem* (CSP) is defined as a triplet $\langle X, D, C \rangle$, where:

- $X = \{x_1, x_2, \ldots, x_n\}$ is a set of n variables
- $D = \{D_1, D_2, \ldots, D_n\}$ is a set of n domains
- $C = \{c_1, c_2, \ldots, c_k\}$ is a set of k constraints
- for each $i \in \{1, \ldots, n\}$, D_i is the domain of the variable x_i

- $n, k \in \mathbb{N}$

We are interested on CSPs where:

- X is a finite set
- D is a finite set
- C is a finite set
- every D_i is a finite and discrete set
- every x_i is an integer variable

Every constraint c_i is defined on a set of variables $V(c_i) \subseteq X$. The constraint c_i restricts the domains of the variables $V(c_i)$. For each i, c_i defines implicity a relation $R(c_i)$ on $V(c_i)$. Such $R(c_i)$ is the set of all the tuples that satisfy the constraint c_i. A constraint is defined by an expression in a certain language.

An example of CSP is the n-queens problem, the problem to place n queens on a n x n chessboard so that they do not threat one another. Assuming that each queen is in a different column, we have a set of n variables, $\{x_1, x_2, \ldots, x_n\}$, and each variable x_i is assigned to the queen in the i^{th} column and indicates the row position of the queen. Obviously each variable x_i has domain $D_i = \{1, 2, \ldots, n\}$. The constraints have to encode the fact that two arbitrary queens cannot be on the same row, column or diagonal. Since we have already assumed that each queen is in a different column, it follows that for each $i, j \in \{1, 2, \ldots, n\}$, such that $i \neq j$, we have:

- $x_i \neq x_j$
- $x_i - x_j \neq i - j$
- $x_j - x_i \neq j - i$

An *assignment* for a CSP is an element of the cartesian product D_1 x D_2 x ... x D_n. An assignment is a *solution* of a CSP if it satisfies all the constraints in C. The set of all the solutions of a CSP is called *solution space.*

A set $K \subseteq C$ is defined *consistent* if and only if K has a solution. K is *inconsistent* if and only if K has no solution.

The *arity* of a constraint is the number of variables it involves. The arity of a CSP is equal to the arity of its highest-arity constraint. Most of the CSPs are binary (with arity two). We call *multiple* CSP a CSP with arity greater than two.

A binary CSP can be represented by a constraint graph (called also constraint network), where the vertices correspond to the CSP variables and the arcs correspond to the CSP constraints. Similarly, a multiple CSP can be instead represented by a constraint hypergraph, where hypervertices correspond to the variables and the hyperedges correspond to the constraints [1] . Two hypervertices are in the same hyperedge if the two related variables occur in a constraint.

A CSP solver is a program that takes in input a certain kind of CSPs and, if the CSP is consistent (has some solutions), returns one or all the solutions of the CSP. This is a decision problem. It is possible also to formulate the optimization version of the problem, where the solver returns only the optimal solution (or the optimal solutions), according to a certain criterion.

Reformulation of multiple CSPs in lower arity CSPs is a common procedure because CSPs of low arity are considerably simpler to treat. Most of the algorithms for solving CSPs restrict to binary CSPs.

Solving a constraint satisfaction problem on a finite domain is an *NP*-complete problem in general. For some special kind of CSPs there are polynomial-time algorithms that solve them.

3.2 Configuration problems

We take into account [3, 4, 5, 6].

The structure of a *configuration problem* derives from the CSP structure. The set of constraints is split into two sets: B, which is called *background* and represents the set of the constraints that must be always satisfied, and F, which is called *foreground* and represents the set of the *dynamic* constraints. Dynamic constraints can be added to B, but the background B must be consistent. If there is a conflict, some dynamic constraints must be relaxed in order to restore the consistency in B. Typically $B \cup F$ is not consistent.

Formally, a configuration problem is defined as $\langle X,\ D,\ B,\ F\rangle$, where:

[1] A *hypergraph* is a generalization of a graph, where edges can connect any number of vertices. Formally, a hypergraph H is a pair $(X,\ E)$ where X is a set of elements, called hypervertices, and E is a set of non-empty subsets of X, called hyperedges. Practically, a hypergraph can be seen as a graph, where an edge can connect more than two vertices. Graphs are hypergraphs where the hyperedges connects two hypervertices.

- X has the same meaning of X for a CSP
- D has the same meaning of D for a CSP
- B is the background
- F is the foreground

In a configuration problem it is defined also a partial order relation [2] on the constraints set (a binary preference relation $\prec$ among the constraints, in order to express the fact that some constraints are more important than other ones). If c_1 and c_2 are constraints, we write $c_1 \prec c_2$ to mean that the satisfaction of c_1 is preferred to the satisfaction of c_2. Sometimes $\prec$ could be a strict partial order relation (every preference defined in the relation $\prec$ involves two distinct constraints). When $\prec$ is a total order relation (all the constraints can be listed in an ordered sequence) we denote it as $<$. A CSP is a decision problem while a configuration problem is an optimization problem. We will see that the criterion to find the optimum is based on the preference relation $\prec$.

In §3.4.1, a simple structure of configuration problem is given using the CP language. Splitting the constraints set into background and foreground, and assigning some preferences between the constraints, we can build a partial order relation $\prec$.

[2] A *partial order* is a binary relation $\trianglelefteq$ over a set G which is reflexive, antisymmetric, and transitive, i.e., for all $a, b, c \in G$, we have that:

- $a \trianglelefteq a$ (reflexivity)
- if $a \trianglelefteq b$ and $b \trianglelefteq a$, then $a = b$ (antisymmetry)
- if $a \trianglelefteq b$ and $b \trianglelefteq c$, then $a \trianglelefteq c$ (transitivity)

A partial order is a *total order* if the binary relation $\trianglelefteq$ is total, i.e., for all $a, b \in G$, we have that:

- $a \trianglelefteq b$ or $b \trianglelefteq a$ (totality)

In a total order, any pair of elements in the set G are mutually comparable under the relation $\trianglelefteq$. A partial order lacks the totality condition, and only some elements are comparable between each other.

A *strict* partial (total) order is a partial (total) order that is not reflexive. In a strict partial (total) order the relation is denoted as $\triangleleft$, to mean the lack of reflexivity.

3.2.1 Preferred relaxations and preferred conflicts

A set $T \subseteq F$ is a *relaxation* of a configuration problem $\langle X, D, B, F\rangle$ if and only if $B \cup T$ has a solution. In other words, a relaxation is a subset of F that is consistent with respect to B. A relaxation exists if and only if B is consistent.
A set $Q \subseteq F$ is a *conflict* of a configuration problem $\langle X, D, B, F\rangle$ if and only if $B \cup Q$ has no solution. In other words, a conflict is a subset of F that is inconsistent with respect to B. A conflict exists if and only if $B \cup F$ is inconsistent.
We assume that the partial order relation $\prec$ among the constraints is always strict. We extend this partial order to a linearization $<$ of $\prec$, which is a strict total order. The linearization $<$ is a superset of $\prec$ and it fulfills the preferences among the constraints. It is a complete specification of this ranking, and it respects the partial order $\prec$, which is incomplete. Typically the partial order $\prec$, and therefore the linearization $<$, are defined only among the dynamic constraints, the constraints in F.

A total order defined on elements can be extended to a total order defined on sets of elements. Hence, in order to compare two relaxations and two conflicts, we introduce two lexicographic extensions of $<$, denoted by $<_{lex}$ and $<_{antilex}$, and defined over sets of constraints. With these two extensions we will be able to compare lexicographically two arbitrary sets of constraints.

Given a strict total order $<$ defined on F, we can enumerate the elements of F in increasing $<$-order $c_1, \ldots, c_n$ (starting with the most important constraints) and compare two subsets X, Y of F lexicographically:

$$X <_{lex} Y$$

$$\Leftrightarrow$$

$$\exists\, j : c_j \in X \setminus Y \text{ and } X \cap \{ c_1, \ldots, c_{j-1} \} = Y \cap \{ c_1, \ldots, c_{j-1} \}$$

A *preferred relaxation* is a relaxation that respects the preference relation $\prec$ and maximizes the selection of the most important constraints. Given a configuration problem $P = \langle X, D, B, F, <\rangle$ where $<$ is a strict total order defined on F, a relaxation T of P is a preferred relaxation of P if and only if there is no other relaxation T^* of P such that $T^* <_{lex} T$.

As we already know, in general the preference relation $\prec$ defined on F is a partial order. In this case a relaxation T of $P = \langle X, D, B, F, \prec\rangle$ is a preferred relaxation of P if and only if there is a linearization $<$ of $\prec$ such that T is a preferred relaxation of $\langle X,\ D,\ B,\ F,\ <\rangle$. In order to compute the preferred relaxation of a configuration problem $P = \langle X,\ D,\ B,\ F,\ \prec\rangle$, first we have to build from $\prec$ a linearization $<$, which is a strict total order.

If $\prec$ is a strict total order and B is consistent, then P has a unique preferred relaxation. Otherwise, in general, different linearizations provide different preferred relaxations.

A preferred relaxation is maximal (non-extensible) because all its proper supersets have no solution. If there are no preferences among the constraints, namely $\prec$ is the empty relation, then the maximal relaxations and the preferred relaxations coincide.

In the same way we define now the preferred conflicts. Given a strict total order $<$ defined on F, we can enumerate the elements of F in increasing $<$-order $c_1, \ldots, c_n$ (starting with the most important constraints) and compare two subsets $X,\ Y$ of F lexicographically in the reverse order:

$$X <_{antilex} Y$$

$$\Leftrightarrow$$

$$\exists\, j :\ c_j \in Y \setminus X \text{ and } X \cap \{\ c_{j+1}, \ldots, c_n\ \} = Y \cap \{\ c_{j+1}, \ldots, c_n\ \}$$

A *preferred conflict* is a conflict that respects the preference relation $\prec$ minimizing the selection of the least important constraints. Given a configuration problem $P = \langle X,\ D,\ B,\ F,\ <\rangle$ where $<$ is a strict total order defined on F, a conflict Q of P is a preferred conflict of P if and only if there is no other conflict Q^* of P such that $Q^* <_{antilex} Q$.

In the general case, when the preference relation $\prec$ defined on F is a strict partial order, a conflict Q of $P = \langle X,\ D,\ B,\ F,\ \prec\rangle$ is a preferred conflict of P if and only if there is a linearization $<$ of $\prec$ such that Q is a preferred conflict of $\langle X,\ D,\ B,\ F,\ <\rangle$. As before, to compute the preferred conflict of a configuration problem $P = \langle X,\ D,\ B,\ F,\ \prec\rangle$, first we have to build a linearization $<$ of $\prec$.

If $\prec$ is a strict total order and $B \cup F$ is inconsistent, then P has a unique preferred conflict. Otherwise, in general, different linearizations provide different preferred conflicts.

A preferred conflict is minimal (irreducible) because all its proper subsets have a solution. If there are no preferences among the constraints, then the minimal conflicts and the preferred conflicts coincide.

There is a strong duality between relaxations and conflicts. Given $X,\ Y \subseteq F$, the relationship between $<_{antilex}$ and $<_{lex}$ is the following:

$$X <_{antilex} Y \Leftrightarrow Y\,(<^{-1})_{lex}\ X$$

If the constraints in F are mutually independent (e.g. have the form of assignments $x_i = v_i$), conflicts correspond to the complements of relaxations of the negated problem with inverted preferences.

Let F be a set of variable-value assignments of the form $x_i = v_i$, and let $\neg c_j \succ^* \neg c_i \Leftrightarrow c_i \prec c_j$, T is a preferred relaxation (conflict) of $P = \langle X,\ D,\ B,\ F,\ \prec\rangle$ if and only if $\{\neg c \mid c \in F \setminus T\}$ is a preferred conflict (relaxation) of $P^* = \langle X,\ D,\ \neg B,\ \{\neg c \mid c \in F\},\ \succ^*\rangle$.

We see now the constructive definitions of preferred relaxations and preferred conflicts. Let $P = \langle X,\ D,\ B,\ F,\ <\rangle$ be a configuration problem such that $<$ is a strict total order defined on F, B is consistent and $B \cup F$ is inconsistent. We enumerate the elements of F in increasing $<$-order $c_1, \ldots, c_n$. The preferred relaxation T of the problem P is T_n, where:

- $T_0 := \emptyset$
- for each $i \in \{1, \ldots, n\}$ we have:

$$T_i = \begin{cases} T_{i-1} \cup \{c_i\} & if\ B \cup T_{i-1} \cup \{c_i\}\ has\ a\ solution \\ T_{i-1} & otherwise \end{cases}$$

The preferred conflict Q of P is constructed in the reversed order and it corresponds to Q_0, where:

- $Q_n := F$
- for each $i \in \{0, 1, \ldots, n\text{-}1\}$ we have:

$$Q_i = \begin{cases} Q_{i+1} \setminus \{c_i\} & if\ B \cup Q_{i+1} \setminus \{c_i\}\ has\ no\ solution \\ Q_{i+1} & otherwise \end{cases}$$

From these definitions we can affirm that the first (with the lowest index) constraint of the $<$-enumeration $c_1, \ldots, c_n$ that is not contained in the preferred relaxation is the last (with the highest index) constraint of the preferred conflict. More formally, given a configuration problem $P = \langle X,\ D,\ B,\ F,\ <\rangle$, where $<$ is a strict total order defined on F, if Q is a preferred conflict of P

and T is a preferred relaxation of P, then the $<$-minimal element of $F \setminus T$ is the $<$-maximal element of Q.

Finding the preferred relaxation and the preferred conflict for a configuration problem P are optimization problems that consider the lexicographic extensions $<_{lex}$ and $<_{antilex}$. In order to compute the preferred relaxation and the preferred conflict of a configuration problem we follow the constructive definitions of preferred relaxation and preferred conflict. We choose an arbitrary linearization $<$ of the preference relation $\prec$, defined on the foreground F, and then we take into account one constraint at a time. We need also a consistency checker which is able to verify whether an arbitrary set of constraints is consistent (has a solution) or not.

We observe now two trivial cases for a configuration problem. If a configuration problem has an inconsistent background B, then there is only one preferred conflict, the empty set, and there are no relaxations. If $B \cup F$ is consistent, then F is the only one preferred relaxation, and there are no conflicts.

In the general case, as discussed previously, in order to find the preferred relaxation T a straightforward algorithm is to start from the background B, and add iteratively (if we don't detect inconsistencies) the constraints of the foreground F in the increasing $<$-order $c_1, \ldots, c_n$. For the preferred conflict Q a straightforward algorithm is to start instead from the whole set $B \cup F$, and we remove iteratively (if we don't find solutions) the constraints of the foreground F in the decreasing $<$-order $c_n, \ldots, c_1$.

3.3 The QuickXplain algorithm

We just saw how to find the preferred relaxation and the preferred conflict for a configuration problem where the dynamic constraints are partially ordered. We have already defined two algorithms that works iteratively. We want now to reduce the number of consistency checks. In order to achieve this goal we will use the strategy to add and remove whole sets of constraints.

Let us split the foreground F into two subsets F_1 and F_2 ($F = F_1 \cup F_2$). If F_1 and F_2 are disjoint ($F_1 \cap F_2 = \emptyset$) and if there are no constraints in F_2 that are preferred to a constraint of F_1, we have that:

- if T_1 is a preferred relaxation of $\langle X, D, B, F_1, \prec\rangle$ and T_2 is a preferred relaxation of $\langle X, D, B \cup T_1, F_2, \prec\rangle$, then $T_1 \cup T_2$ is a preferred relaxation of $\langle X, D, B, F_1 \cup F_2, \prec\rangle$.

- if Q_2 is a preferred conflict of $\langle X,\ D,\ B \cup F_1,\ F_2,\ \prec\rangle$ and Q_1 is a preferred conflict of $\langle X,\ D,\ B \cup Q_2,\ F_1,\ \prec\rangle$, then $Q_1 \cup Q_2$ is a preferred conflict of $\langle X,\ D,\ B,\ F_1 \cup F_2,\ \prec\rangle$.

We introduce now the QuickXplain algorithm. This algorithm computes the preferred relaxation and the preferred conflict for a configuration problem with dynamic constraints partially ordered. It uses the above properties and we will show it in two different versions, one for the preferred relaxation, and the other for the preferred conflict. The algorithm is recursive and uses the divide-and-conquer paradigm. QuickXplain works by recursively breaking down a problem into two sub-problems of the same type, until these become simple enough to be solved directly.

QuickXplain is a meta-algorithm, parametric on two functions: *isConsistent()* and *split()*.

The function *isConsistent()* is a consistency checker on a set of constraints. It returns *true* if the set of constraints has solutions, otherwise it returns *false*.

The function *split()* gives a positive number smaller than n, and can be chosen in different ways. In order to exploit in the best way the properties of QuickXplain, we will divide the foreground F in two smaller sets of the same size, choosing *split(n)* $:= n/2$.

The input of this algorithm is a configuration problem with a background B, a foreground F, and a strict partial order $\prec$ defined on F, which expresses some preferences among the foreground constraints. As we discussed before, in order to compute preferred relaxations and preferred conflicts we need a strict total order $<$, an extension of $\prec$ that enumerates all the constraints in F. The QuickXplain algorithm chooses arbitrarily $<$.

For our configuration problems we consider only strict partial orders and strict total orders. We do not allow a constraint to be in relation with itself. In fact, the binary relation $\prec$, defined among the foreground constraints, expresses the preference between two constraints and the reflexivity of $\prec$ makes no sense (a constraint can not be preferred to itself).

In the implementation of QuickXplain, a directed graph is used for computing the linearization. Every vertex of the graph represents a different foreground constraint, and a directed arc between two vertexes represents the fact that the constraint of the first vertex is preferred to the constraint of the second vertex. The transitive property of $\prec$ ensures that, for any path in the graph, the constraints of the first vertexes are preferred respect the constraints represented by the vertexes at the end of the path.

The linearization algorithm returns the total order, the enumeration of all the constraints, as follows. The first constraints of the enumeration correspond to two different kinds of constraint: constraints that are not in relation with any other constraints, and the most preferred ones (constraints that are preferred to other constraints, but no other constraints can be preferred to them). Depending on a user setting, the algorithm can returns these first constraints respecting the same order in which they are stored or in a different shuffled order. Then, all the other constraints are enumerated. The algorithm uses a queue which stores the vertexes to enumerate. At the beginning all the "first" constraints are added to the queue. Then the algorithm traverses the preferences graph in breadth-first search (BFS), and it appends [3] at the end of the queue only those vertexes having no incoming edges from vertexes not yet included in the queue. The enumeration of the constraints is obtained listing the elements of the queue, from the first to the last.

It is important to observe that for each step the foreground F is divided into two disjoint sets, F_1 and F_2, where, due to the linearization $<$, all the constraints contained in F_1 are more important to any constraint of F_2. Hence, we can use the properties stated above, applying the divide-and-conquer strategy.

Let us start first with the preferred conflict version of QuickXplain. Initially all the constraints (background B and foreground F) are passed to the consistency checker. If they are all satisfied, the whole configuration problem is consistent and there are no conflicts.

If instead $B \cup F$ is inconsistent, we try to remove some dynamic constraints. At each step the foreground F is split in two smaller sets: F_1 and F_2. There are two recursive calls where the problem size is simpler. In the first one it is used $B \cup F_1$ as background and F_2 as foreground, while in the second one it is used $B \cup Q_2$ as background and F_1 as foreground. The first call returns Q_2 as preferred conflict, while the second one returns Q_1.

Using the divide-and-conquer technique we reduce the number of consistency checks. When we split the foreground F in F_1 and F_2, if the half problem with F_2 as foreground has an inconsistent background $(B \cup F_1)$, then we can remove all the dynamic constraints contained in F_2 with only a single consistency check [4] . Otherwise, if the current background $B \cup F_1$ is consistent,

[3] The queue does not contains duplicates. A vertex is added to the queue only if the queue does not contain it.

[4] The parameter Δ represents the set of dynamic constraints that the previous iteration of QuickXplain has added to the background B. When $\Delta = \emptyset$, no constraints are added to B in the previous iteration, and therefore we skip the consistency check because the algorithm has already called the method *isConsistent()* on that consistent background B.

we try to re-add some dynamic constraints that are in F_2.

The recursion stops when the foreground contains only one constraint or when the background is not consistent. In these cases we do not split again the foreground. At each step, foreground and background change, the dynamic constraints move from the foreground to the background and back.

The output, which is built step by step after the recursive calls ($Q_1 \cup Q_2$), is the preferred conflict for a configuration problem with a background B, a foreground F, and a preferences partial order $\prec$ defined on F.

3.3.1 QuickXplain pseudocode for the preferred conflict

Algorithm 3.3.1: *QuickXplain_C*(B, F, $\prec$)

```
if $isConsistent(B \cup F)$
  then return ($no\ conflicts$)
  else { if $F = \emptyset$
           then return ($\emptyset$)
           else return ($QuickXplain\_C^*(B,\ B,\ F,\ \prec)$)
```

Algorithm 3.3.2: *QuickXplain_C**(B, Δ, F, $\prec$)

```
if $\Delta \neq \emptyset$
  then { if $\neg isConsistent(B)$
           then return ($\emptyset$)
  else if $F = \{c\}$
  then return ($\{c\}$)
  else { let $\{\alpha_1, \alpha_2, \ldots, \alpha_n\}$ an enumeration of $F$ that extends $\prec$
         comment: a strict total order that extends the partial order $\prec$
         $k := split(n)$ ($with\ k \in \{1, \ldots, n-1\}$)
         comment: we will choose the function $split(n) := n/2$
         $F_1 := \{\alpha_1, \ldots, \alpha_k\}$
         $F_2 := \{\alpha_{k+1}, \ldots, \alpha_n\}$
         comment: $F$ is split in 2 smaller sets: $F_1$ and $F_2$
         $Q_2 := QuickXplain\_C^*(B \cup F_1,\ F_1,\ F_2,\ \prec)$
         $Q_1 := QuickXplain\_C^*(B \cup Q_2,\ Q_2,\ F_1,\ \prec)$
         return ($Q_1 \cup Q_2$)
```

We see now the other version of QuickXplain, for computing the preferred relaxation. It works in a similar way of the previous version, using the duality between relaxations and conflicts.

Initially we check the consistency of the background B. If some constraints in B are not satisfied, there is a conflict in B (the background B has no solution) and the configuration problem has no relaxations.

Otherwise, B is consistent, and therefore we try to add some dynamic constraints to B. As before, at each step the foreground F is split in two smaller sets: F_1 and F_2, and the algorithm is recursively called on the two subproblems.

In the first call the background is still B and the foreground becomes F_1, while in the second call the background is $B \cup T_1$ and the foreground is F_2. The first call returns T_1 as preferred relaxation, while the second one returns T_2.

Here, when we split the foreground F in F_1 and F_2, if the first half problem is such that all the constraints in the background and in the foreground are satisfied ($B \cup F_1$ is consistent), then we can add to B all the dynamic constraints contained in F_1 with only a single consistency check [5] . Otherwise, we try to remove some dynamic constraints that are in F_1.

The recursion stops when the foreground contains only one constraint or when $B \cup F$ is a consistent set. In these cases we do not split again the foreground.

The output is the preferred relaxation for a configuration problem with a background B, a foreground F, and a preferences partial order $\prec$ defined on F.

[5] This time we introduced a new parameter, Ω, which is the foreground in the calling iteration. The parameter Δ still represents the incremental addition of dynamic constraints to the background B. When $\Delta \cup F = \Omega$, the previous iteration has added all the foreground constraints of the first subproblem, hence we skip the consistency check because the algorithm has already called the method *isConsistent()* on that set $B \cup F$.

3.3.2 QuickXplain pseudocode for the preferred relaxation

Algorithm 3.3.3: *QuickXplain_R*$(B,\ F,\ \prec)$

```
if $\neg isConsistent(B)$
  then return (no relaxations)
  else { if $F = \emptyset$
           then return $(\emptyset)$
           else return $(QuickXplain\_R^*(B,\ B,\ F,\ F,\ \prec))$
```

Algorithm 3.3.4: *QuickXplain_R*$^*(B,\ \Delta,\ F,\ \Omega,\ \prec)$

```
if $\Delta \cup F \neq \Omega$
  then { if $isConsistent(B \cup F)$
           then return $(F)$
  else if $F = \{c\}$
  then return $(\emptyset)$
  else { let $\{\alpha_1, \alpha_2, \ldots, \alpha_n\}$ an enumeration of $F$ that extends $\prec$
         comment: a strict total order that extends the partial order $\prec$
         $k := split(n)$ (with $k \in \{1, \ldots, n-1\}$)
         comment: we will choose the function $split(n) := n/2$
         $F_1 := \{\alpha_1, \ldots, \alpha_k\}$
         $F_2 := \{\alpha_{k+1}, \ldots, \alpha_n\}$
         comment: $F$ is split in 2 smaller sets: $F_1$ and $F_2$
         $T_1 := QuickXplain\_R^*(B,\ \emptyset,\ F_1,\ F,\ \prec)$
         $T_2 := QuickXplain\_R^*(B \cup T_1,\ T_1,\ F_2,\ F,\ \prec)$
         return $(T_1 \cup T_2)$
```

3.4 The CLab library

CLab [6] is an open source C++ library for fast backtrack-free interactive product configuration.

Here, a problem describes a product, and it consists of two parts: a set of variables with finite domains denoting the product free parameters, and a set of rules (the constraints) defining the legal product configurations.

[6]This chapter draws on the CLab 1.0 user manual [9]

We are not interested on how CLab works or how it is made. We are only interested on the problems syntax required by CLab. This language is called CP.

3.4.1 The CP language

We explain now the CP language, a language to define problems. Here we make two simplifications on the language (for simplicity we will call it also CP): we will deal with only one kind of expressions, the logical expressions, and we will take into account only natural numbers. In the original definition of CP language, the user can define also arithmetical expressions and it is possible to use all the integer numbers.

CP has three types: Boolean, range and enumeration. A range is a consecutive and finite sequence of natural numbers. An enumeration is a finite set of values, where a single value is like an identifier: a sequence of digits, capital letters, small letters and underscore characters ‘_’ that must not begin with a digit. Range and enumeration are useful to define user types.

CP implicitly defines the Boolean type (bool) which is a range type from 0 to 1, where 0 represents *false* and 1 represents *true*. A CP description consists of a type declaration, a variable declaration, and a rule declaration. The type declaration is optional if no range or enumeration types are defined. Square brackets indicate an optional part, with the only exception of a range type declaration, that requires explicitly ‘[’ and ‘]’. A natural number is of course a sequence of digits. A rule is an expression, namely a combination of round brackets, identifiers, numbers and operators.

Here is the complete CP syntax:

cp ::= [type { **typedecl** }] variable { **vardecl** } rule { **ruledecl** }

typedecl ::= identifier [number . . number] ; [**typedecl**]
| identifier { **idlst** } ; [**typedecl**]

vardecl ::= **vartype idlst** ; [**vardecl**]

vartype ::= bool
| identifier

idlst ::= identifier [, **idlst**]

```
ruledecl ::= exp ; [ ruledecl ]

exp      ::= number
          |  identifier
          |  ! exp
          |  ( exp )
          |  exp op exp

op       ::= >>| || | && | == | != | !
```

The semantics of the operators is defined as in C/C++ language. Hence, !, ==, !=, &&, and || denote logical negation, equality, inequality, conjunction, and disjunction, respectively. The only exception is the operator >> that denotes logical implication.

The only unary operator is the negation. Equality, inequality and implication are binary operators, while conjunction and disjunction can have two or more operands.

Precedence order and associativity of the operators are defined in this way:

Operators	Associativity
!	right to left
!= ==	left to right
&&	left to right
||	left to right
>>	left to right

If another order is required, parentheses can be used in the conventional way.

In order to become familiar with this syntax, we show now a simple problem written in CP language. A school gives the possibility to the students to learn a foreign language in the afternoon or in the night. Four possible languages can be chosen: Japanese, Chinese, Korean and Russian. Unfortunately at the moment no Chinese teachers have been found by the school, hence only Japanese, Korean and Russian language are available. Each student must attend one and only one language course. For each language, the students can choose the level of difficulty of the course. Five possible levels have been thought by the school, but only the first three levels are available to the students.

Hence, in CP we have three types to enumerate languages, levels, time, and the correspondent three variables to express the choices of the students. The rules describe formally the constraints of the problem. Here is the CP syntax for the example above:

```
type
    language {Japanese, Chinese, Korean, Russian};
    level [1 .. 5];
    when {morning, afternoon, night};

variable
    language lang;
    level le;
    when w;

rule
    (le == 1 || le == 2 || le == 3);
    lang != Chinese;
    w != morning;
```

3.5 Related work

There exist several commercial tools to solve constraint satisfaction problems and configuration problems. There are also free libraries that allow to develop configuration or constraint-based applications. Some of these tools/libraries are:

- ILOG [54], which includes many commercial tools that solve and manage complex CSPs and configuration problems
- Configit Product Modeler [55], a shareware tool for developing customized configurator applications
- Choco [56], a free software architecture for variables domains, constraints, propagation and tree search
- Comet [57], a free tool for solving complex combinatorial optimization problems in areas such as resource allocation and scheduling

- JaCoP [58], a Java library for modeling and solving specific CSP domains

- Cream [59], a Java library useful to develop programs requiring constraint satisfaction or optimization on finite domains

- Koalog [60], a commercial tool for solving CSPs and complex problems in areas such as scheduling, routing, configuration

- CLab [8], a C++ library for fast backtrack-free interactive product configuration

- Gecode [61], a C++ library for developing constraint-based systems and applications

Part II

Original contributions

Chapter 4

Solving CSPs via SAT encoding

In order to implement the QuickXplain algorithm we need a consistency checker (see the function *isConsistent()* in §3.3), which is able to verify whether an arbitrary set of constraints is consistent (has a solution) or not. The first idea that comes to mind is obviously to use a CSP solver as consistency checker. Although there exists free CSP solvers (see §3.5), no free solvers have been found for the chosen CP problems. However, a CSP can be transformed into SAT. In fact, given a CSP $\langle X,\ D,\ C\rangle$, each constraint $c_i \in C$ can be encoded into a propositional formula φ_{c_i}, and the set of all the constraints C becomes the conjunction of all the correspondent constraints formulas: $\varphi_{c_1} \wedge \varphi_{c_2} \wedge \cdots \wedge \varphi_{c_n}$.

A propositional formula can be converted in CNF and a SAT solver can check whether it is satisfiable or not. Hence, the CSP $\langle X,\ D,\ C\rangle$ is consistent (has solutions) if and only if $\varphi_{c_1} \wedge \varphi_{c_2} \wedge \cdots \wedge \varphi_{c_n}$ is satisfiable. In the same way, the set $K = \{c_j, c_{j+1}, \ldots, c_k\} \subseteq C$ is consistent if and only if the propositional formula $\varphi_{c_j} \wedge \varphi_{c_{j+1}} \wedge \cdots \wedge \varphi_{c_k}$ is satisfiable.

In this chapter we will explain in details how to encode a CSP in a propositional formula. First of all the rules of the constraints are converted using only Boolean variables and then a CNF conversion is made.

4.1 The Boolean encoding

As explained in §3.4.1, a problem written in CP language can have variables of only three possible types: Boolean, range or enumeration. The range type has a domain of n consecutive natural numbers, while an enumeration domain is composed in general by n different alphanumeric values. The first step of our Boolean encoding is to convert every non-Boolean domain of n elements in the set 0, 1, ..., n-1, realizing a bijective map, a correspondence, from the original domain to the first n natural numbers, and vice-versa.

The second step is to represent every value in the set 0, 1, ..., n-1 using only Boolean values.

Every natural number k>1 can be written in binary representation using $\lceil \log_2(k) \rceil$ bits [1] , namely a sequence of $\lceil \log_2(k) \rceil$ digits in {0, 1}.

Following this idea, if a non-Boolean domain D needs $y = \lceil \log_2(|D|) \rceil$ [2] bits to encode its elements, every value $d \in D$ can be represented in binary as an array of Boolean values $\boldsymbol{d} = (d_{y-1}, \dots, d_1, d_0)$. Analogously, every non-Boolean variable x which has domain D can be encoded by an array of Boolean variables $\boldsymbol{x} = (x_{y-1}, \dots, x_1, x_0)$.

Thus, the formula $x = d$ can be represented as the Boolean formula $x_{y-1} = d_{y-1} \wedge \dots \wedge x_1 = d_1 \wedge x_0 = d_0$, which can be written also as $\boldsymbol{x} = \boldsymbol{d}$, and the formula $x \neq d$ can be represented as the Boolean formula $x_{y-1} \neq d_{y-1} \vee \dots \vee x_1 \neq d_1 \vee x_0 \neq d_0$, which can be written also as $\boldsymbol{x} \neq \boldsymbol{d}$. In the real implementation, each sub-formula $x_i = d_i$ is simplified in one of the two atomic sub-formulas x_i and $\neg x_i$, depending on the Boolean value of d_i, that can be respectively 1 or 0. Analogously, we simplified every $x_i \neq d_i$ as $\neg x_i$ or x_i.

In CP language, the only operators that involve non-Boolean operands are equalities (==) and inequalities (!=). But now we know how to convert such expressions into Boolean formulas. Hence, since logical negation (!), conjunction (&&), disjunction (||) and logical implication (>>) are defined only among Boolean formulas, any CP expression can be rewritten as a Boolean formula.

It is important to observe that this encoding does not guarantee surjectivity. In particular, the encoding is surjective only when the domain D to encode has a cardinality $|D|$ which is a multiple of the number two. In fact, if the domain D contains k values (namely, $|D|$ = k), then the Boolean variables introduced by the encoding can represent the set of the first $2^{\lceil \log_2(k) \rceil}$ natural numbers, whose cardinality is in general greater than k. Hence, the encoding introduces Boolean variables that can represent non-allowed values [3] . The

[1] The $\lceil\ \rceil$ function, called "ceiling", maps every number to the next higher integer. Formally, $\lceil x \rceil = \min\{n \in \mathbb{Z} \mid n \geq x\}$.

[2] In this context, the notation $|\ |$ represents the cardinality of a set, which is the number of its elements.

[3] As before, we consider only the case of k>1. When $k = 0$, the domain D is empty and we do not need to encode it. When $k = 1$, we use only one bit to represent D, and

number of these extra-values are $2^{\lceil \log_2(k) \rceil} - k$. For example, a domain of five elements is encoded using three bits ($\lceil \log_2(5) \rceil = 3$), but three bits encode the first eight natural numbers ($2^3 = 8$).

Therefore, in our implementation the Boolean encoding adds a "domain constraint" that forbids the $2^{\lceil \log_2(k) \rceil} - k$ unwanted combinations.

Overall, in order to encode a CSP into a Boolean formula we have to:

1. convert the domain of each non-Boolean variable in a set of consecutive natural numbers
2. introduce an array of Boolean variables for each non-Boolean variable
3. add the domain constraint
4. replace all the equalities and inequalities among non-Boolean operands (variables and/or values) in each expression defined in the constraints

For example, the domain {*Monday, Tuesday, Wednesday, Thursday, Friday*} is first converted in {0, 1, 2, 3, 4}. Then, using three bits, all the variables and the values of this domain are encoded. Here is the binary encoding of the domain values:

original value	number	binary encoding
Monday	0	000
Tuesday	1	001
Wednesday	2	010
Thursday	3	011
Friday	4	100

The domain constraint forbids the values that are not allowed in the domain. Here are the values that are not allowed:

number	binary encoding
5	101
6	110
7	111

we have one non-allowed value.

Due to the domain constraint, it is possible to convert any expression in a Boolean formula and vice-versa: from any Boolean formula, knowing the domain of the variables involved in the formula, it is possible to go back to the original non-Boolean formula. This will be useful when we have to find the solution of the original problem from a solution of the correspondent encoded propositional formula, returned by a SAT solver.

4.2 CNF conversion and incremental SAT solving

We have seen how to represent a problem using only Boolean variables and Boolean values. The next step is to convert in CNF the Boolean formulas representing the constraints expressions. This step is fundamental because MiniSat, the SAT solver used, accepts only formulas in CNF. In the implementation the labeling CNF conversion is used. In particular the first algorithm explained in (§2.3.2) is implemented.

All the constraints are converted in CNF Boolean formulas. Checking the consistency of a constraints set is equivalent to verify if the related Boolean formulas are satisfiable. When the QuickXplain algorithm is called (§3.3), every consistency check (function *isConsistent()*) calls MiniSat on the conjunction of the CNF Boolean formulas converted from the constraints rules that are selected.

The tool can be set to use classic or incremental SAT solving. By default it uses MiniSat incrementally.

From §4.1 and §2.3.2 we know that every constraint c_i of the configuration problem has an associated CNF Boolean formula φ_{c_i}, that includes in general more than one clause. During the CNF conversion new Boolean variables are introduced in order to represent the labels needed by the CNF conversion algorithm, and every φ_{c_i} is stored for each constraint c_i in two representations: the classic one with Boolean variables, operators and parentheses, and that one in DIMACS format.

If QuickXplain is called using the classic SAT solving option, for every consistency check on a set of constraints $K = \{c_j, c_{j+1}, \dots, c_k\}$, the correspondent formulas $\varphi_{c_j}, \varphi_{c_{j+1}}, \dots, \varphi_{c_k}$ are selected, and the clauses of the formula $\varphi_{c_j} \wedge \varphi_{c_{j+1}} \wedge \cdots \wedge \varphi_{c_k}$ are written in DIMACS format in a textfile. Hence, MiniSat checks the satisfiability of the Boolean formula written in the textfile.

When the tool is set to the incremental SAT solving option, a C++ program interfaces with MiniSat, which is used with the mechanism of the assumptions. As seen in §2.5, incremental SAT solving is useful when we want to solve the satisfiability of a set of related formulas $\{\phi_1, \phi_2, \ldots, \phi_n\}$. In this modality MiniSat stores a general formula from which is possible, passing some assumptions to the solver, to check the satisfiability of the formulas in $\{\phi_1, \phi_2, \ldots, \phi_n\}$.

In our case, after the creation of a solver instance, we add to the solver all the Boolean variables and all the clauses of the formula $\varphi_{c_1} \wedge \varphi_{c_2} \wedge \cdots \wedge \varphi_{c_n}$, which represents the problem with all its constraints. In order to check the consistency of a certain constraint c_i we need to say to the solver to check the formula φ_{c_i}.

From the labeling CNF conversion we know that there is a Boolean variable b_i for each formula φ_{c_i}. Since the CNF conversion adds the clause $b_i \Leftrightarrow \varphi_{c_i}$ for each constraint c_i, b_i is a label for the whole formula φ_{c_i} (and hence for the constraint c_i), and it is *true* if and only if φ_{c_i} is *true*. In order to select some constraints, we just need to assign *true* to the Boolean variables that label the constraints to select. This mechanism corresponds to make the assumptions in MiniSat.

For example, in order to check the consistency of the constraint c_j we pass the assumption b_j, and to check the consistency of the set $\{\varphi_{c_j} \wedge \varphi_{c_{j+1}} \wedge \cdots \wedge \varphi_{c_k}\}$ we have to assume $b_j \wedge b_{j+1} \wedge \cdots \wedge b_k$. Every time we want to solve a formula we need to reset from the solver the old assumptions and insert the new ones.

In order to perform incremental SAT solving we need to remove, for each constraint c_i, the first clause of the associated formula φ_{c_i}. The first clause of φ_{c_i} corresponds to the Boolean variable b_i (unit clause) that labels the whole formula φ_{c_i}. This is done because at the beginning all the CNF Boolean formulas of all the constraints must be added to the SAT solver, but no conditions on the constraints consistency must be specified. The assumptions are useful to select the constraints. In fact, when QuickXplain checks the consistency on a certain set of constraints, it first selects the first clauses (the labels) of the constraints involved, and it adds to the SAT solver the assumptions. These assumptions say that these Boolean variables must be *true*, and correspond to say that some constraints must be satisfied. Before any check consistency, all the previous assumptions are deleted and new assumptions are added.

In order to understand better, let's see again the example §2.3.1. From the formula $b_1 \wedge CNF^*((\neg x_1 \vee b_2) \Leftrightarrow b_1) \wedge CNF^*((x_2 \wedge x_3 \wedge x_4 \wedge b_3) \Leftrightarrow b_2) \wedge CNF^*((x_5 \Leftrightarrow x_6) \Leftrightarrow b_3)$ we have to remove the first clause, b_1. Now the formula $CNF^*((\neg x_1 \vee b_2) \Leftrightarrow b_1) \wedge CNF^*((x_2 \wedge x_3 \wedge x_4 \wedge b_3) \Leftrightarrow b_2) \wedge$

$CNF^*((x_5 \Leftrightarrow x_6) \Leftrightarrow b_3)$ can be added to the SAT solver. When we want to check the satisfiability of this formula, we first add the assumption b_1 to the SAT solver that simplifies the formula on that assumption. It is important to notice that all the formulas are already added in the SAT solver, adding and removing the assumptions correspond to select and deselect formulas from the solver.

Chapter 5

The tool

5.1 Technologies used

NetBeans [51] has been used as integrated development environment (IDE) to develop the system. The system is implemented in Java [50] programming language. There is also a small C++ program which uses the C source code of MiniSat 2.0 [10]. This program manages directly the variables, the clauses and the assumptions used internally by MiniSat and is invoked by the tool using Java Native Interface (JNI), a framework that allows Java applications to call and be called by native applications and libraries written in other languages, such as C, C++ and Assembly.

Since Java applications are multi-platform, to run the tool in a certain operating system you need to recompile the C++ program and build a shared native library that allows the JNI bridge between the Java Virtual Machine (JVM) and the C++ process.

The tool has been tested in Ubuntu [48] 8.04 (Hardy Heron), a free operating system based on Debian [49], a popular Linux distribution.

The application is written using Java 6 Standard Edition, and other two Java libraries: ANTLR [52] and JGraphT [53]. ANTLR is a lexer and parser generator that uses LL parsing. It generates a lexical analyzer (lexer) and a syntactic analyzer (parser) from a grammar file. The lexical analyzer scans a string and return it as a sequence of tokens that are used by the syntactic analyzer to produce a parse tree which represents the syntactic structure of the string according to the formal rules defined in the grammar.

JGraphT is instead a free Java class library that provides mathematical graph-theory objects and algorithms. JGraphT supports various types of graphs. In the implementation simple directed graphs are used.

5.2 Architecture of the system

We show now the architecture of the realized system; first of all an overview of the components (Figure §5.1), and then a detailed description for each component (Figure §5.2).

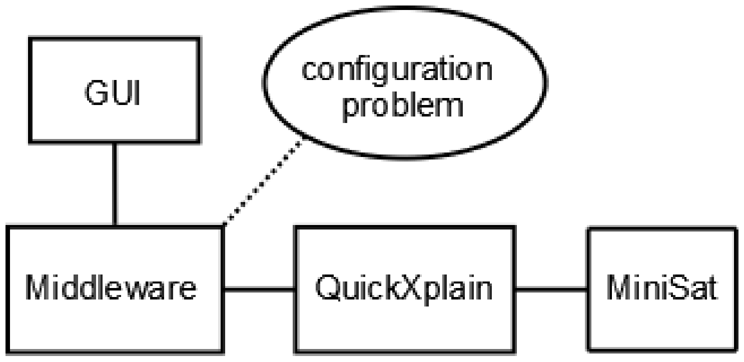


Figure 5.1: General architecture

On the highest level there is a Graphical user interface (GUI) which interacts with the user. The GUI is built on top of a middleware. This level is responsible to represent internally the configuration problem and call the QuickXplain algorithm. It provides many functions to the user in order to manage the configuration problems. There is also another module, the implementation of the QuickXplain algorithm, needed to compute the outputs to the user. This module calls the MiniSat solver to check the consistency of the constraints sets. The input is a configuration problem $\langle X, D, B, F\rangle$, with the foreground F partially ordered, and the outputs are the preferred conflict Q and the preferred relaxation T, with a solution for the set $B \cup T$.

The whole system is realized in Java, a part the program that interfaces to MiniSat, realized in C++. Using the GUI, the user can load a problem, which is taken from CLib (Configuration Benchmarks Library) [7]. Every problem is written in CP language and is composed by a set of variables and a set of constraints. The latter set is for each benchmark a consistent set (has solutions) and it represents the background of the configuration problem. The foreground constraints and the preferences among them can be loaded, saved and/or can be added and modified using the GUI.

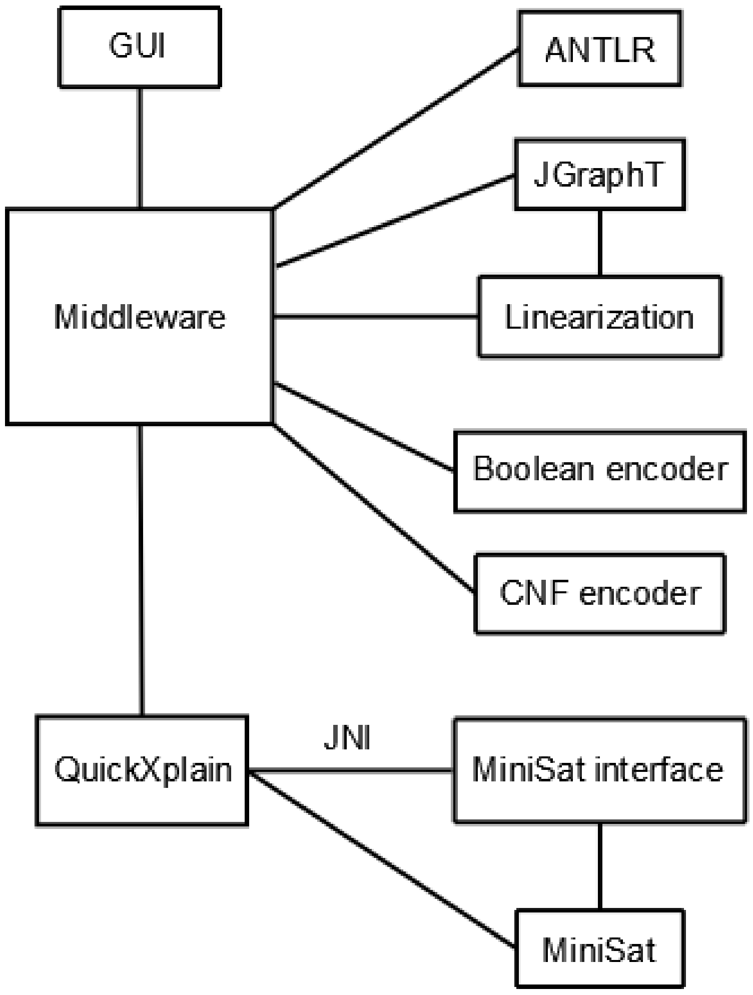


Figure 5.2: Detailed architecture

The middleware uses the ANTLR tool to scan and parse all the constraints (contained in background and in the foreground) and the JGraphT library to represent every constraint expression as a tree. There is a module, the Boolean encoder, that encodes all the benchmark variables in Boolean variables and rewrites all the constraints expressions as Boolean formulas, using the new Boolean variables. The Boolean encoder implements the algorithm explained is §4.1.

Hence, each constraint is a Boolean formula that can be translated in conjunctive normal form. This task is done by the CNF encoder, another module of the middleware. This module implements what is explained in §4.2.

The Linearization module collects the preferences between the foreground constraints, a partial order set, and extends it in a total order, an ordered sequence of all the foreground constraints. This module has been developed using JGraphT library and uses a directed graph to store the preferences between constraints.

When the user wants to compute the preferred relaxation or the preferred conflict of a configuration problem, the middleware calls the QuickXplain algorithm, module that needs MiniSat in order to check the consistency of sets of constraints. MiniSat can be called in two ways, directly as a process (in the case of classic SAT solving option), or using JNI and a C++ interface for the MiniSat solver (in the case of incremental SAT solving).

5.3 Benchmarks used

The benchmark problems used are taken from CLib (Configuration Benchmarks Library) [7]. CLib offers many benchmarks written in four different languages.

We took into account problems written in CP language that have no constraints defined with arithmetical operators.

These benchmarks have a list of variables and a list of constraints. There are no restrictions on the arity of the constraints. Since every benchmark is consistent, all the constraints are classified as background constraints. The foreground constraints are added by the user and are required to make the whole problem, or better, the set of all the constraints, as inconsistent. The user has also to insert some preferences between the foreground constraints, defining a strict partial order relation on the foreground.

5.4 Functionalities

The tool allows the user to:

- load a specific configuration problem, which is composed by a benchmark (the benchmark describes the variables and the background of the problem), a foreground and a set of preferences among the foreground constraints
- insert, modify, delete temporally or permanently, constraints in the foreground

Figure 5.3: The main window

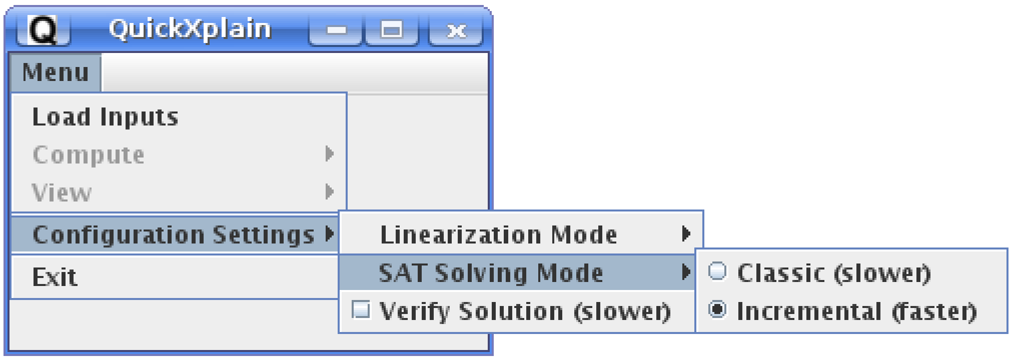
- insert, modify, delete temporally or permanently, preferences among the foreground constraints
- extend the preferences partial order to a total order
- set the SAT solving mode: classic or incremental
- set the way how the linearization algorithm computes the total order: ordered or shuffled
- compute the preferred conflict, the preferred relaxation and the related solution, for a configuration problem
- verify effectively that a solution of MiniSat corresponds to a solution of the problem
- save foreground, preferences, and the outputs (conflicts, relaxations and solutions) to files

We will explain in details these functionalities.

5.5 GUI

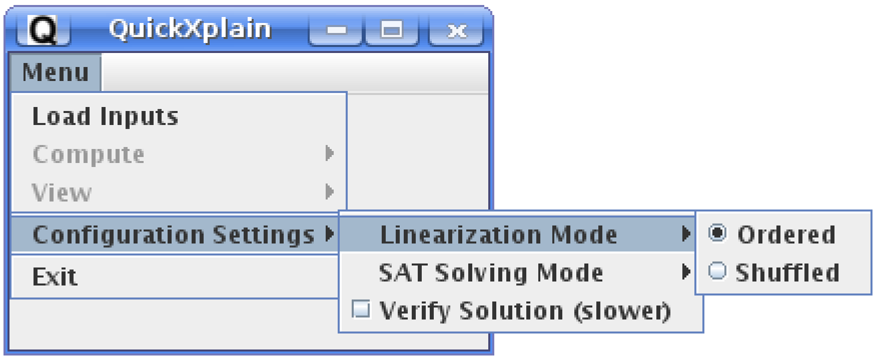
5.5.1 A typical user session

At the beginning the user sees the main window of the tool. The user may need to set some configurations. It is possible to enable the verification of the solution (this option is disabled by default), choose the SAT solving mode



○ Classic (slower)
● Incremental (faster)

Figure 5.4: Setting the SAT solving mode



● Ordered
○ Shuffled

Figure 5.5: Setting the linearization mode

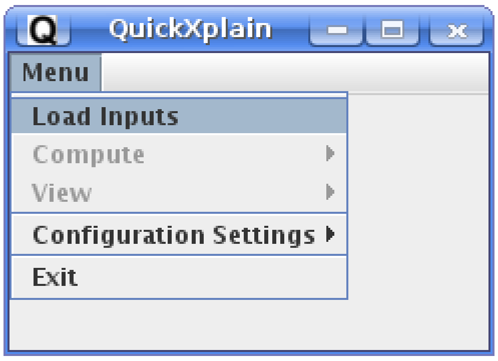


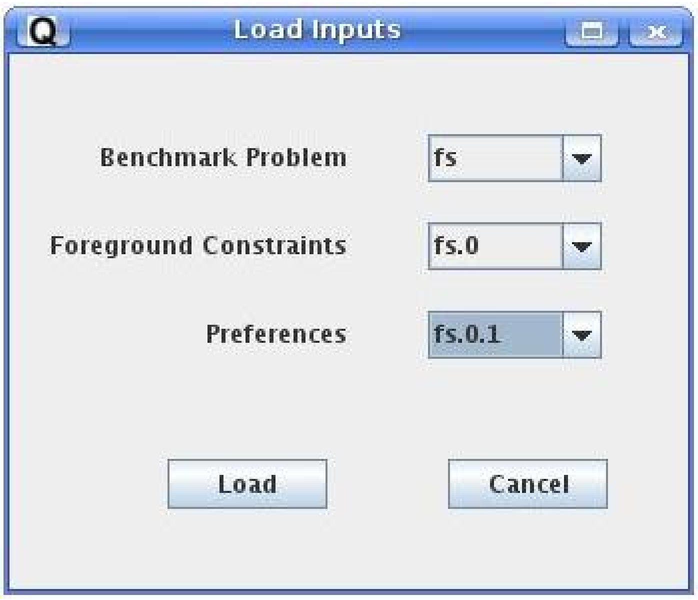
Figure 5.6: Loading a configuration problem

and decide the option about the linearization algorithm (enumerate the first constraints in an ordered or shuffled sequence). At anytime these settings can be changed.

A part the settings, the first operation is to load the configuration problem. Once the “Load Inputs” item is selected, a new window will appear. From this window the user can choose the benchmark, the foreground and its preferences. While a benchmark needs to be chosen, foreground and preferences can be left empty. In Figure §5.7 the user chooses the benchmark “fs” with non-empty foreground and preferences. Once the configuration problem has been loaded, four windows will appear in the screen:

- the variables window
- the background constraints window
- the foreground constraints window
- the preferences window

The variables window shows, for each variable of the benchmark, the name, the type and the values allowed. The background and foreground constraints windows show respectively the expressions of the background and foreground constraints. In the preferences window there is the list of all the preferences among the foreground constraints. The variables and the background constraints windows do not allow the user to modify the rows of the table. In fact, the tool forbids the user to modify the benchmark. At the contrary, the foreground and the preferences windows provide to the user many functionalities to manage and modify the foreground constraints and their preferences. Foreground constraints can be added, modified, removed temporally (see the

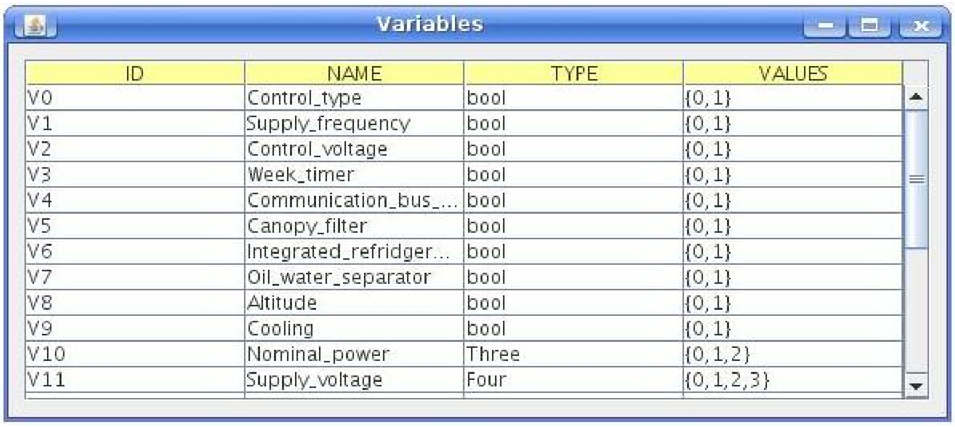


Benchmark Problem: fs
Foreground Constraints: fs.0
Preferences: fs.0.1

Figure 5.7: Loading “fs” configuration problem

“Hide” button) or permanently, and saved to files (the user can later reload the same foreground). The same operations can be done on the preferences. Alterations on the foreground update the preferences structure: for example, if the user removes a foreground constraint, the preferences on this constraint will be removed. Figure §5.12 shows a foreground where the user has removed temporally some constraints, while Figure §5.13 shows the window to insert a preference (it appears when the user presses the “Insert” button on the preferences window).

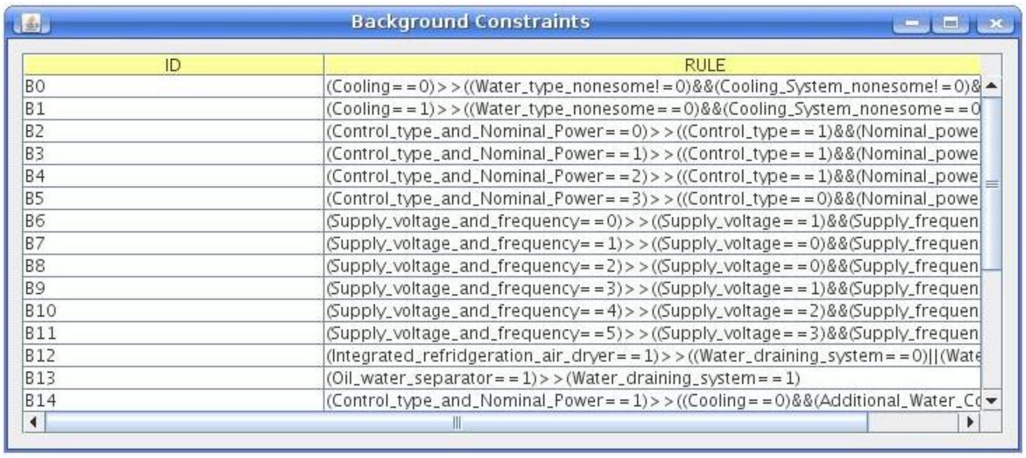
The configuration problem is loaded, and the user can perform operations on the foreground and on the preferences. Pressing the “Compute” item, the user can choose what to compute, the preferences total order, the preferred conflict or the preferred relaxation. A new window will show the result of the computation that we have asked. The preferences total order is computed running the linearization algorithm. The algorithm extends the preferences partial order and returns the list of all the foreground constraints in a new order. If we alter the preferences or the foreground structure, the linearization algorithm returns in general a different preferences total order. This operation is useful to see which total order QuickXplain will use to compute the preferred relaxation and the preferred conflict. When the user presses the specific items to compute the preferred conflict or the preferred relaxation, the tool first computes the total order (because the user can anytime alter the foreground or the preferences structure) and then it calls the QuickX-



Variables

ID	NAME	TYPE	VALUES
V0	Control_type	bool	{0,1}
V1	Supply_frequency	bool	{0,1}
V2	Control_voltage	bool	{0,1}
V3	Week_timer	bool	{0,1}
V4	Communication_bus_...	bool	{0,1}
V5	Canopy_filter	bool	{0,1}
V6	Integrated_refridger...	bool	{0,1}
V7	Oil_water_separator	bool	{0,1}
V8	Altitude	bool	{0,1}
V9	Cooling	bool	{0,1}
V10	Nominal_power	Three	{0,1,2}
V11	Supply_voltage	Four	{0,1,2,3}

Figure 5.8: Benchmark variables window

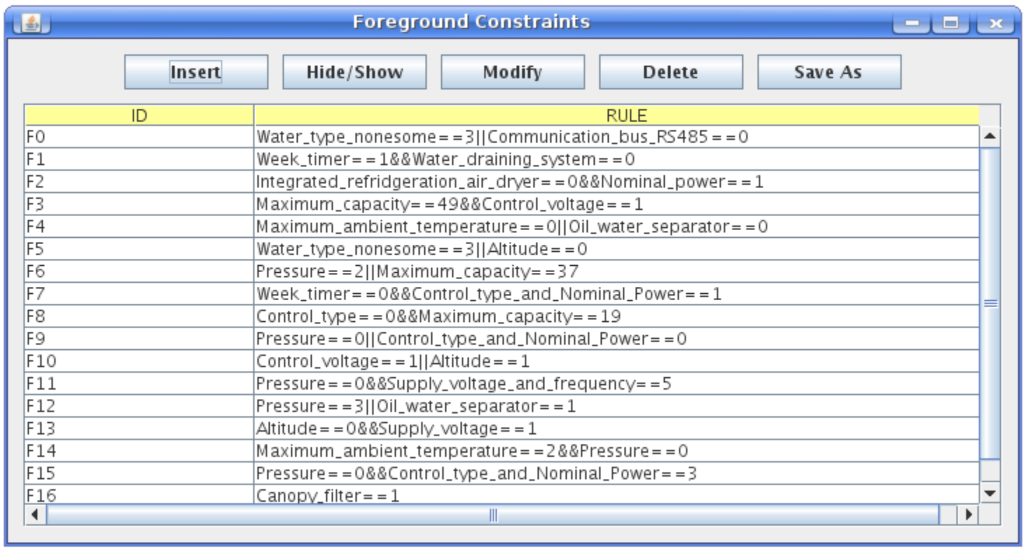


Background Constraints

ID	RULE
B0	(Cooling==0)>>((Water_type_nonesome!=0)&&(Cooling_System_nonesome!=0)&
B1	(Cooling==1)>>((Water_type_nonesome==0)&&(Cooling_System_nonesome==0
B2	(Control_type_and_Nominal_Power==0)>>((Control_type==1)&&(Nominal_powe
B3	(Control_type_and_Nominal_Power==1)>>((Control_type==1)&&(Nominal_powe
B4	(Control_type_and_Nominal_Power==2)>>((Control_type==1)&&(Nominal_powe
B5	(Control_type_and_Nominal_Power==3)>>((Control_type==0)&&(Nominal_powe
B6	(Supply_voltage_and_frequency==0)>>((Supply_voltage==1)&&(Supply_frequen
B7	(Supply_voltage_and_frequency==1)>>((Supply_voltage==0)&&(Supply_frequen
B8	(Supply_voltage_and_frequency==2)>>((Supply_voltage==0)&&(Supply_frequen
B9	(Supply_voltage_and_frequency==3)>>((Supply_voltage==1)&&(Supply_frequen
B10	(Supply_voltage_and_frequency==4)>>((Supply_voltage==2)&&(Supply_frequen
B11	(Supply_voltage_and_frequency==5)>>((Supply_voltage==3)&&(Supply_frequen
B12	(Integrated_refridgeration_air_dryer==1)>>((Water_draining_system==0)||(Wate
B13	(Oil_water_separator==1)>>(Water_draining_system==1)
B14	(Control_type_and_Nominal_Power==1)>>((Cooling==0)&&(Additional_Water_Co

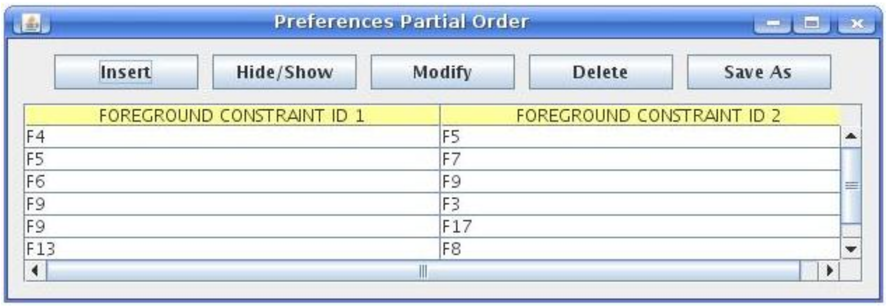
Figure 5.9: Background constraints window

plain algorithm. In the case of the preferred relaxation, the set of constraints composed of the background and the preferred relaxation is a consistent set, and the tool will also show a window with the solution (list of the value assignments to the benchmark variables) of this constraints set. The solution is found computing the values to the original variables from the values of the new Boolean variables introduced to encode the problem into SAT, values that form the model given by MiniSat. The user can anytime hide or show the tool windows, selecting them from the "View" item or closing the related windows.



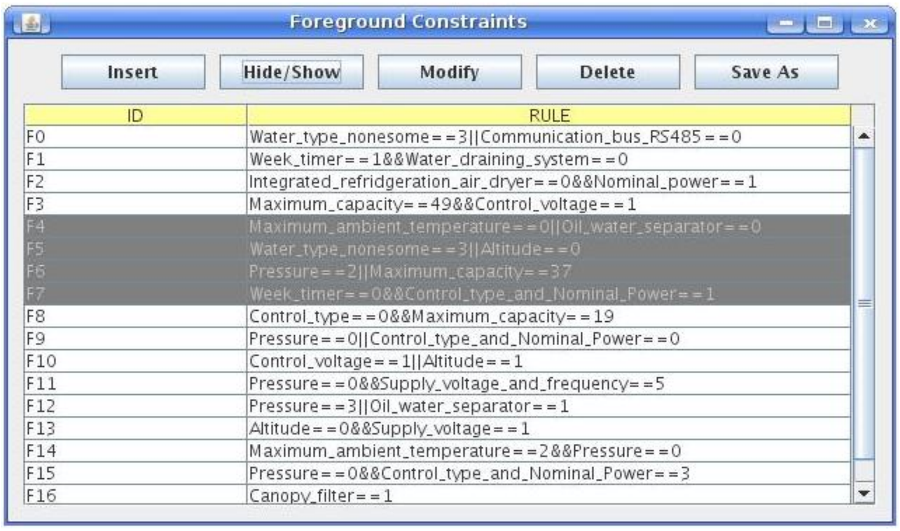
ID	RULE
F0	Water_type_nonesome==3||Communication_bus_RS485==0
F1	Week_timer==1&&Water_draining_system==0
F2	Integrated_refridgeration_air_dryer==0&&Nominal_power==1
F3	Maximum_capacity==49&&Control_voltage==1
F4	Maximum_ambient_temperature==0||Oil_water_separator==0
F5	Water_type_nonesome==3||Altitude==0
F6	Pressure==2||Maximum_capacity==37
F7	Week_timer==0&&Control_type_and_Nominal_Power==1
F8	Control_type==0&&Maximum_capacity==19
F9	Pressure==0||Control_type_and_Nominal_Power==0
F10	Control_voltage==1||Altitude==1
F11	Pressure==0&&Supply_voltage_and_frequency==5
F12	Pressure==3||Oil_water_separator==1
F13	Altitude==0&&Supply_voltage==1
F14	Maximum_ambient_temperature==2&&Pressure==0
F15	Pressure==0&&Control_type_and_Nominal_Power==3
F16	Canopy_filter==1

Figure 5.10: Foreground constraints window



FOREGROUND CONSTRAINT ID 1	FOREGROUND CONSTRAINT ID 2
F4	F5
F5	F7
F6	F9
F9	F3
F9	F17
F13	F8

Figure 5.11: Preferences window



ID	RULE
F0	Water_type_nonesome==3||Communication_bus_RS485==0
F1	Week_timer==1&&Water_draining_system==0
F2	Integrated_refridgeration_air_dryer==0&&Nominal_power==1
F3	Maximum_capacity==49&&Control_voltage==1
F4	Maximum_ambient_temperature==0||Oil_water_separator==0
F5	Water_type_nonesome==3||Altitude==0
F6	Pressure==2||Maximum_capacity==37
F7	Week_timer==0&&Control_type_and_Nominal_Power==1
F8	Control_type==0&&Maximum_capacity==19
F9	Pressure==0||Control_type_and_Nominal_Power==0
F10	Control_voltage==1||Altitude==1
F11	Pressure==0&&Supply_voltage_and_frequency==5
F12	Pressure==3||Oil_water_separator==1
F13	Altitude==0&&Supply_voltage==1
F14	Maximum_ambient_temperature==2&&Pressure==0
F15	Pressure==0&&Control_type_and_Nominal_Power==3
F16	Canopy_filter==1

Figure 5.12: Example of foreground with some "hidden" constraints

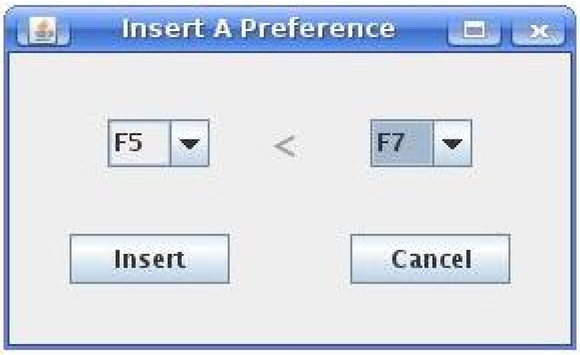


Figure 5.13: Inserting a preference between two foreground constraints

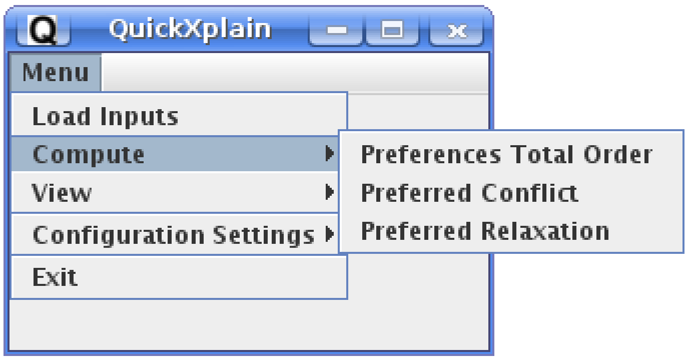
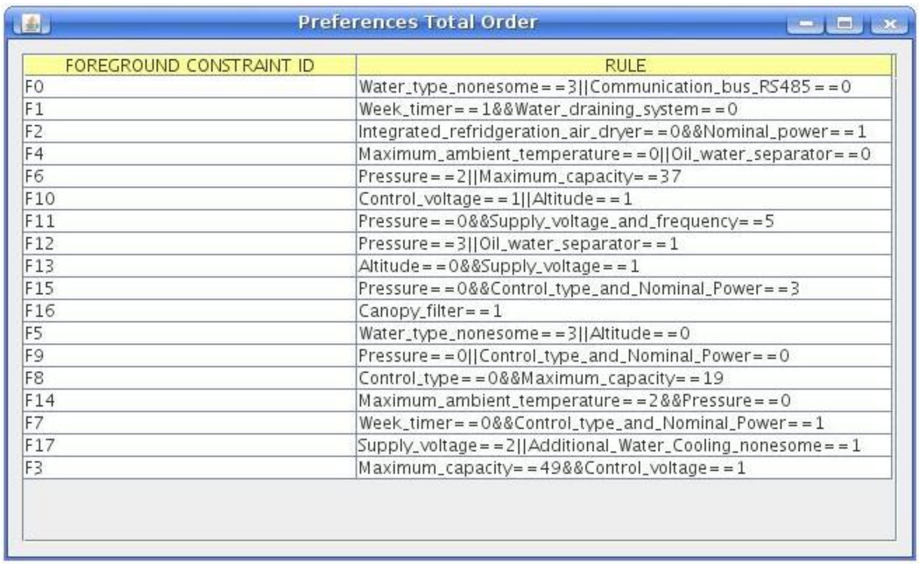
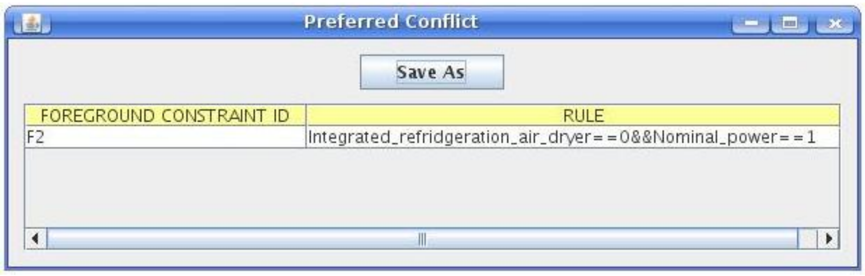


Figure 5.14: Computing something



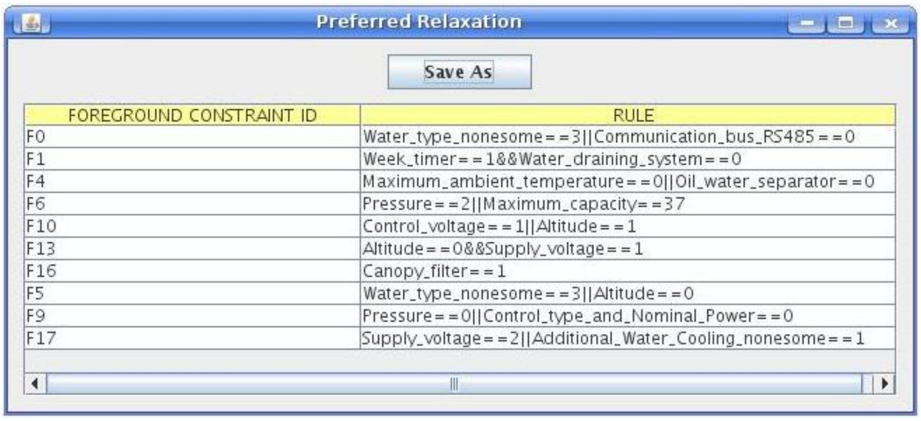
FOREGROUND CONSTRAINT ID	RULE
F0	Water_type_nonesome==3||Communication_bus_RS485==0
F1	Week_timer==1&&Water_draining_system==0
F2	Integrated_refridgeration_air_dryer==0&&Nominal_power==1
F4	Maximum_ambient_temperature==0||Oil_water_separator==0
F6	Pressure==2||Maximum_capacity==37
F10	Control_voltage==1||Altitude==1
F11	Pressure==0&&Supply_voltage_and_frequency==5
F12	Pressure==3||Oil_water_separator==1
F13	Altitude==0&&Supply_voltage==1
F15	Pressure==0&&Control_type_and_Nominal_Power==3
F16	Canopy_filter==1
F5	Water_type_nonesome==3||Altitude==0
F9	Pressure==0||Control_type_and_Nominal_Power==0
F8	Control_type==0&&Maximum_capacity==19
F14	Maximum_ambient_temperature==2&&Pressure==0
F7	Week_timer==0&&Control_type_and_Nominal_Power==1
F17	Supply_voltage==2||Additional_Water_Cooling_nonesome==1
F3	Maximum_capacity==49&&Control_voltage==1

Figure 5.15: The preference total order returned by the linearization algorithm



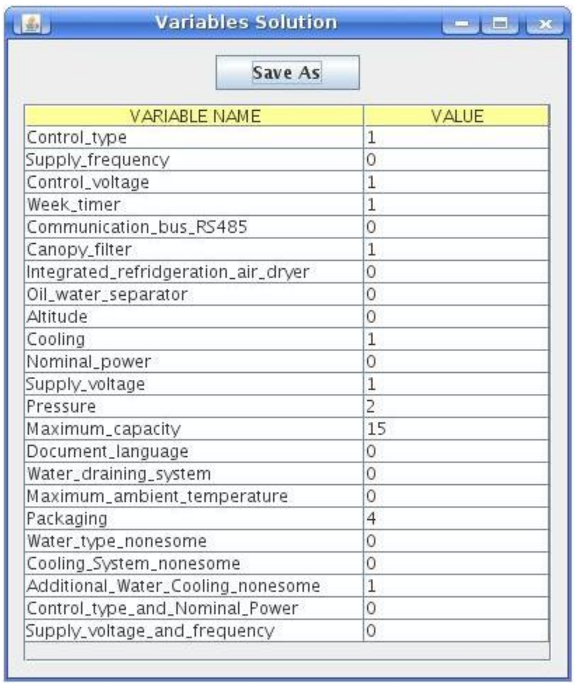
FOREGROUND CONSTRAINT ID	RULE
F2	Integrated_refridgeration_air_dryer==0&&Nominal_power==1

Figure 5.16: The preferred conflict



FOREGROUND CONSTRAINT ID	RULE
F0	Water_type_nonesome==3||Communication_bus_RS485==0
F1	Week_timer==1&&Water_draining_system==0
F4	Maximum_ambient_temperature==0||Oil_water_separator==0
F6	Pressure==2||Maximum_capacity==37
F10	Control_voltage==1||Altitude==1
F13	Altitude==0&&Supply_voltage==1
F16	Canopy_filter==1
F5	Water_type_nonesome==3||Altitude==0
F9	Pressure==0||Control_type_and_Nominal_Power==0
F17	Supply_voltage==2||Additional_Water_Cooling_nonesome==1

Figure 5.17: The preferred relaxation



VARIABLE NAME	VALUE
Control_type	1
Supply_frequency	0
Control_voltage	1
Week_timer	1
Communication_bus_RS485	0
Canopy_filter	1
Integrated_refridgeration_air_dryer	0
Oil_water_separator	0
Altitude	0
Cooling	1
Nominal_power	0
Supply_voltage	1
Pressure	2
Maximum_capacity	15
Document_language	0
Water_draining_system	0
Maximum_ambient_temperature	0
Packaging	4
Water_type_nonesome	0
Cooling_System_nonesome	0
Additional_Water_Cooling_nonesome	1
Control_type_and_Nominal_Power	0
Supply_voltage_and_frequency	0

Figure 5.18: Variables solution

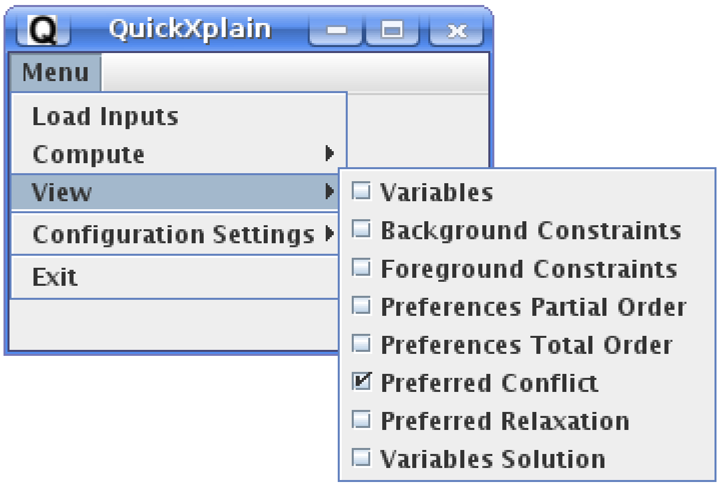


Figure 5.19: To view some windows

Chapter 6

Empirical results

We performed a simulation on the selected benchmark problems using a Pentium Dual-Core 1.60 GHz machine with 2GB of memory, running Linux Ubuntu Hardy Heron. Here we show the obtained results. A script generated fifteen different sets of foreground constraints for each problem and fifteen sets of preferences for each foreground. More precisely, each foreground has a random number of constraints, whose formulas are composed by a finite number of variables, operators ad values. These elements are chosen randomly with respect to the domain of the benchmark variables. In the same way, preferences between the foreground constraints are generated randomly. In this way we had two hundred twenty-five (fifteen times fifteen) instances for each benchmark problem. Each instance is composed by a background, a foreground and a set of preferences. Hence we had two hundred twenty-five runs for each configuration problem.

In the table §6.1 every row summarizes the statistics for a distinct configuration problem. The columns of the table are:

- benchmark: the name of the benchmark problem
- bgs: the number of background constraints
- fgs: the average number of foreground constraints
- pfs: the average number of preferences among the foreground constraints
- cf: the average number of foreground constraints contained in the preferred conflict

- %cf: the average percentage of foreground constraints contained in the preferred conflict
- rx: the average number of foreground constraints contained in the preferred relaxation
- %rx: the average precentage of foreground constraints contained in the preferred relaxation
- time: the average of the total execution times in milliseconds (times required to parse and encode an instance of the problem, and solve it finding its outputs) [1]

Moreover, for each instance of the configuration problems, we ran the tool using both the incremental SAT solving and the classic SAT solving. We took into account the total times required by MiniSat to solve all the encoded formulas (time needed to perform all the calls to the method *isConsistent()*) incrementally and in the classic way. Hence, we made a comparison between these two different ways to use the SAT solver. We drew a scatter plot for each problem. In the plot, each problem instance is represented by a point, whose coordinates are the times in milliseconds spent by MiniSat using the classic SAT solving (x-axis) and the incremental SAT solving (y-axis). Figures §6.1 and §6.2 show the scatter plots for the benchmarks "chinese" and "fs". In the appendix §A we report the scatter plots for all the configuration problems.

Looking into the results we can affirm that the execution time grows with respect to the constraints complexity: problems with many constraints, and problems with constraints described by long formulas, are complicated to solve. For the reasons explained in §2.5 and §4.2, the incremental SAT solving modality is generally faster respect the classic SAT solving. Another fact to observe is that while preferred conflicts have only one or two foreground constraints, preferred relaxations contain at least the 65% of the foreground constraints.

[1]Since incremental SAT solving is quicker than classic SAT solving, the execution times are computed running MiniSat incrementally

benchmark	bgs	fgs	pfs	cf	%cf	rx	%rx	time
baobab3	107	68	26	2	2.94%	49	72.06%	807
chinese	36	38	16	2	5.26%	25	65.79%	305
das9201	82	90	39	2	2.22%	66	73.33%	1247
das9202	36	45	19	2	4.44%	32	71.11%	382
das9203	30	45	18	1	2.22%	33	73.33%	366
das9204	30	37	14	1	2.70%	28	75.68%	322
das9205	20	35	13	1	2.86%	28	80.00%	291
das9206	112	109	41	1	0.92%	79	72.48%	1674
das9207	324	291	127	2	0.69%	201	69.07%	11396
das9208	145	136	54	2	1.47%	91	66.91%	2584
das9209	73	86	37	1	1.16%	65	75.58%	1102
edf9201	132	129	51	2	1.55%	89	68.99%	2510
edf9202	435	434	176	2	0.46%	290	66.82%	27661
edf9203	475	381	163	2	0.52%	241	63.25%	22372
edf9204	375	331	131	2	0.60%	219	66.16%	15246
edf9205	142	193	76	2	1.04%	133	68.91%	4896
edf9206	362	319	131	1	0.31%	220	68.97%	13265
edfpa14b	290	235	99	1	0.43%	162	68.94%	8378
edfpa14o	173	270	113	2	0.74%	196	72.59%	9406
edfpa14p	101	112	46	2	1.79%	79	70.54%	1984
edfpa14q	194	279	109	2	0.72%	200	71.68%	10341
edfpa14r	132	115	48	2	1.74%	79	68.70%	2077
edfpa15b	249	254	105	2	0.79%	174	68.50%	8892
edfpa15o	138	232	95	2	0.86%	171	73.71%	7148
edfpa15p	80	106	45	2	1.89%	76	71.70%	1664
edfpa15q	158	257	106	2	0.78%	192	74.71%	8532
edfpa15r	110	88	39	2	2.27%	61	69.32%	1700
elf9601	242	190	84	2	1.05%	125	65.79%	4913
esvs	32	12	5	1	8.33%	9	75.00%	220
fs	25	14	6	1	7.14%	9	64.29%	201
ftr10	94	100	36	1	1.00%	78	78.00%	1387
isp9602	122	106	43	2	1.89%	75	70.75%	1671
isp9603	95	94	36	2	2.13%	66	70.21%	1324
isp9604	132	176	72	2	1.14%	125	71.02%	4146
isp9606	41	75	29	2	2.67%	57	76.00%	876
isp9607	65	60	26	2	3.33%	45	75.00%	623
jbd9601	315	351	146	2	0.57%	260	74.07%	17164

Table 6.1: Statistics for all the benchmark problems

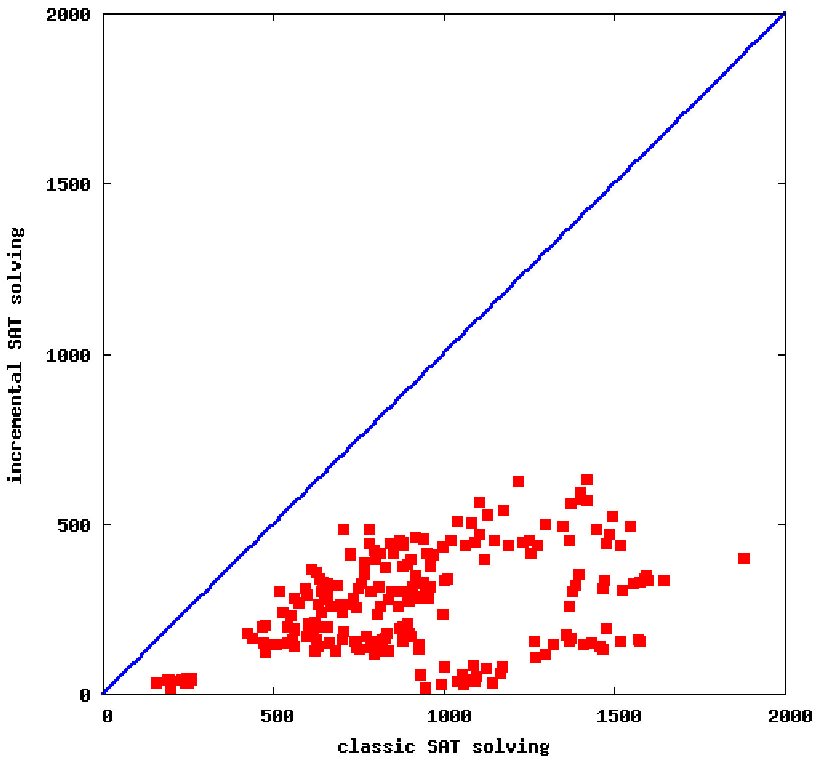


Figure 6.1: “chinese” benchmark

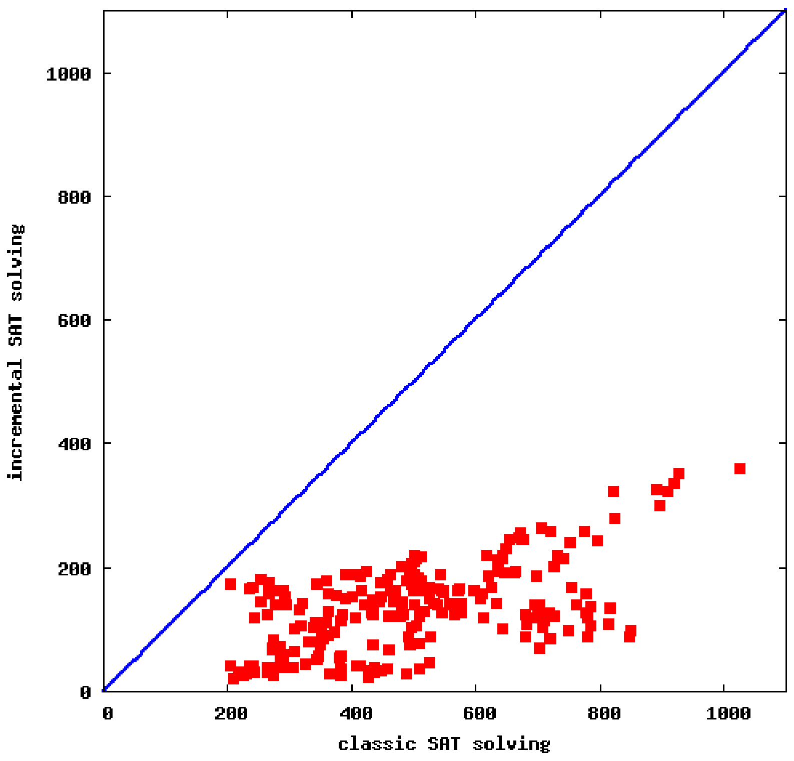


Figure 6.2: “fs” benchmark

Chapter 7

Conclusions

We have explained here a possible way to solve configuration problems, by implementing the QuickXplain algorithm and using a SAT encoding approach. The problems we solved are taken from the Configuration Benchmarks Library (CLib) [7].

The realized tool implements the QuickXplain algorithm and computes the preferred conflict and the preferred relaxation for configuration problems on which is defined a binary preference relation among the constraints, in order to express the fact that some constraints are more important than other ones. MiniSat, an efficient SAT solver, is used to solve the satisfiability of the encoded formulas (propositional formulas that represent the constraints rules in the SAT encoding), and represents the consistency checker of the QuickXplain algorithm.

Appendix A

Comparison of time spent by classic and incremental SAT solving

Here there are the scatter plots for all the configuration problems. Every plot shows the problem instances as points, whose coordinates are the times in milliseconds spent by MiniSat using the classic SAT solving (x-axis) and the incremental SAT solving (y-axis).

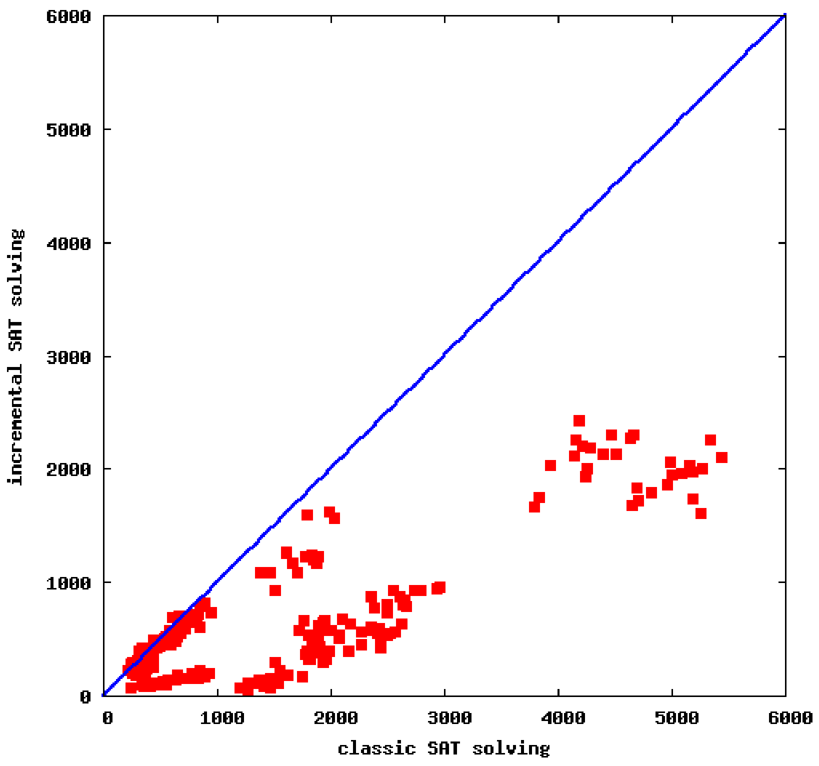


Figure A.1: “baobab3” benchmark

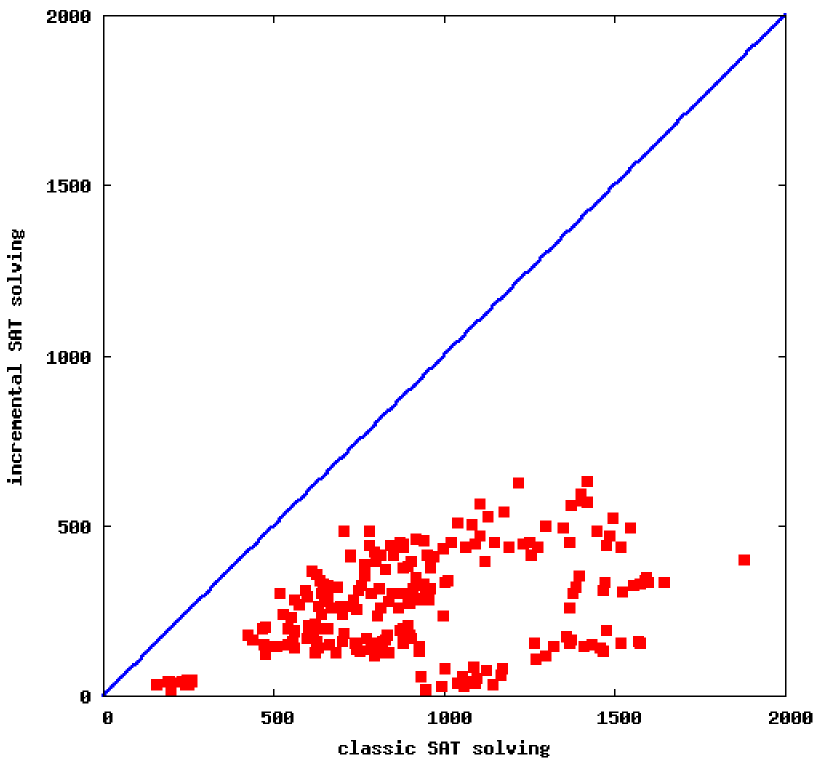


Figure A.2: “chinese” benchmark

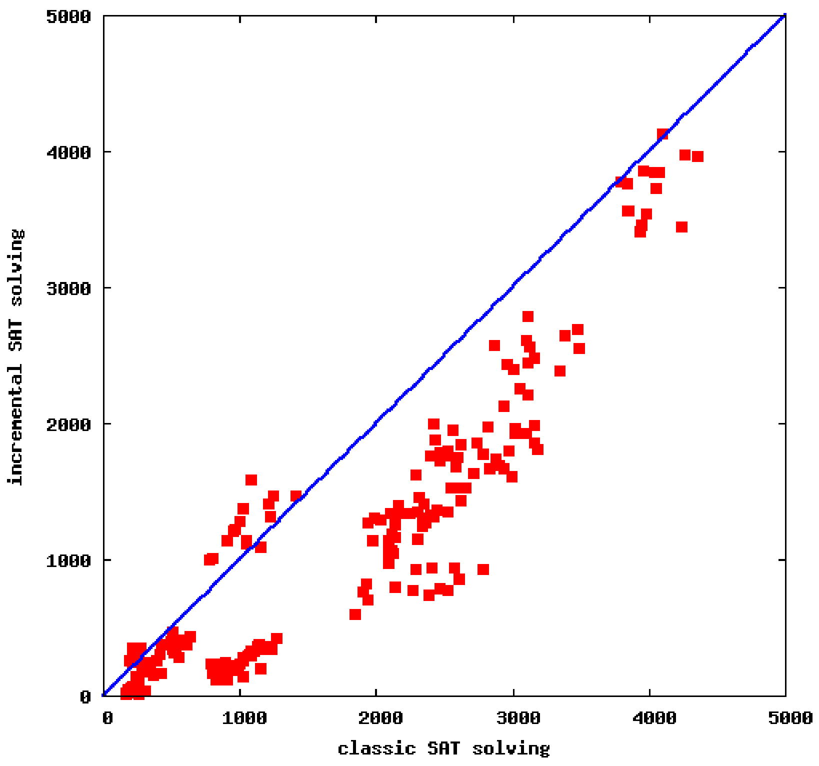


Figure A.3: “das9201” benchmark

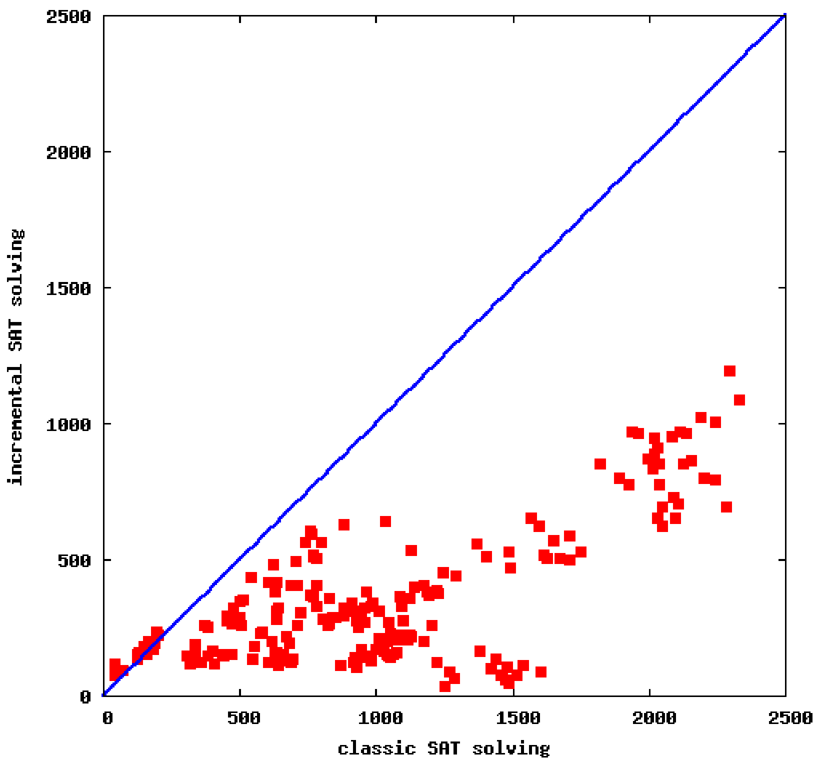


Figure A.4: “das9202” benchmark

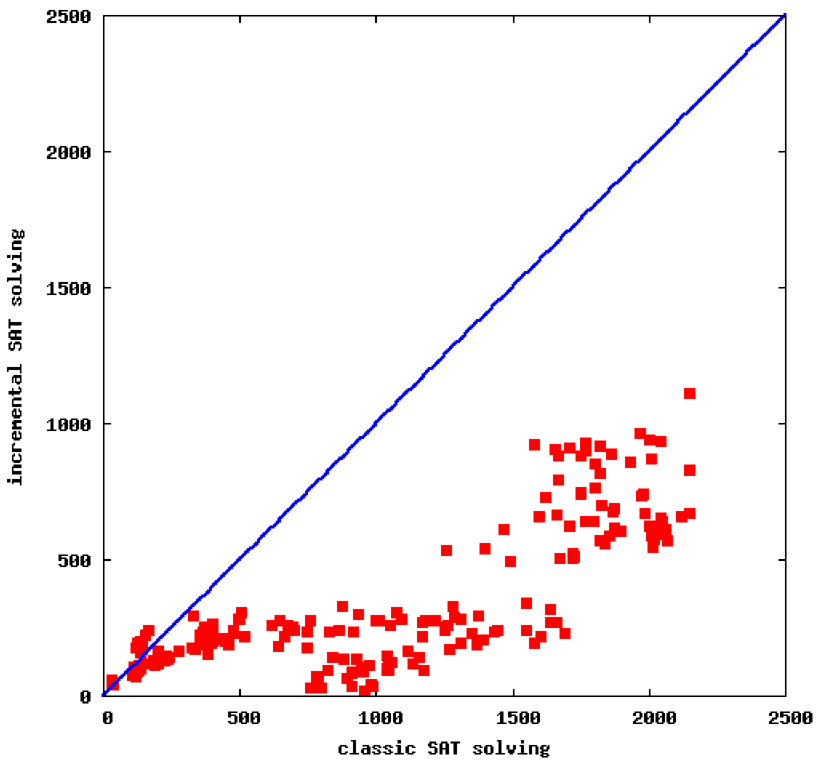


Figure A.5: “das9203” benchmark

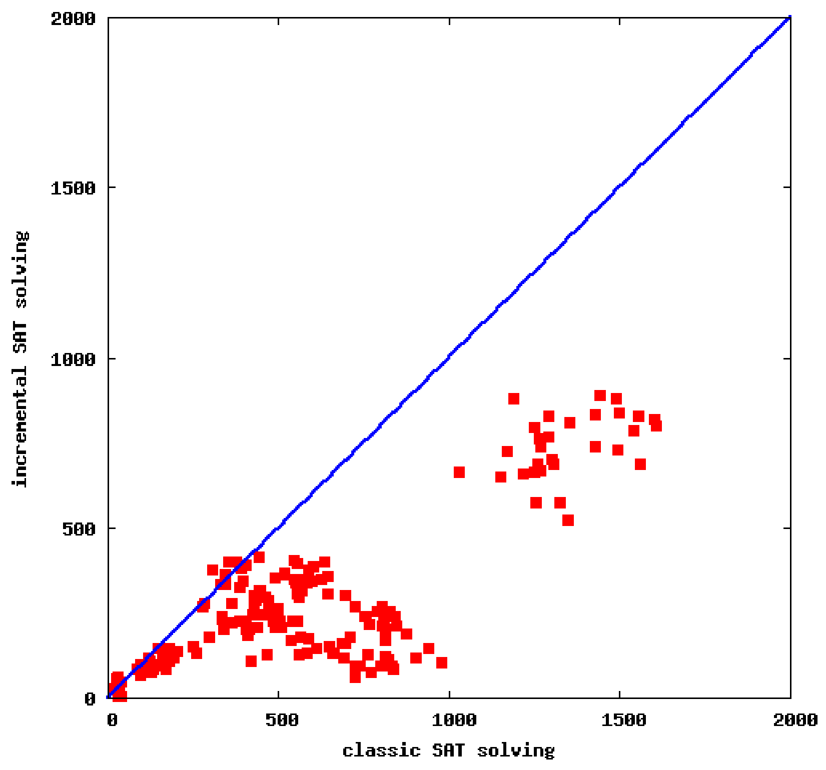


Figure A.6: “das9204” benchmark

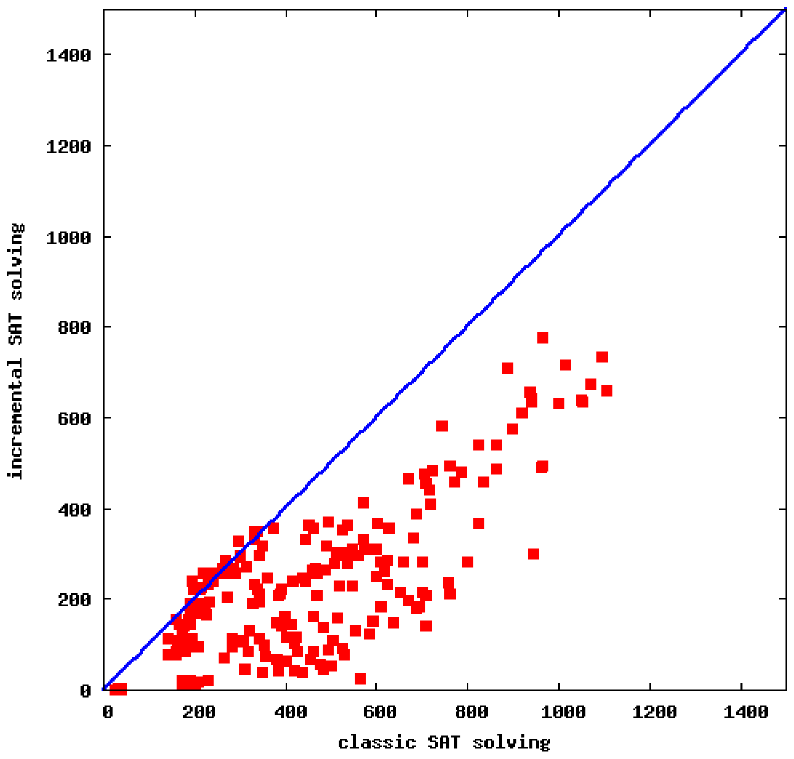


Figure A.7: “das9205” benchmark

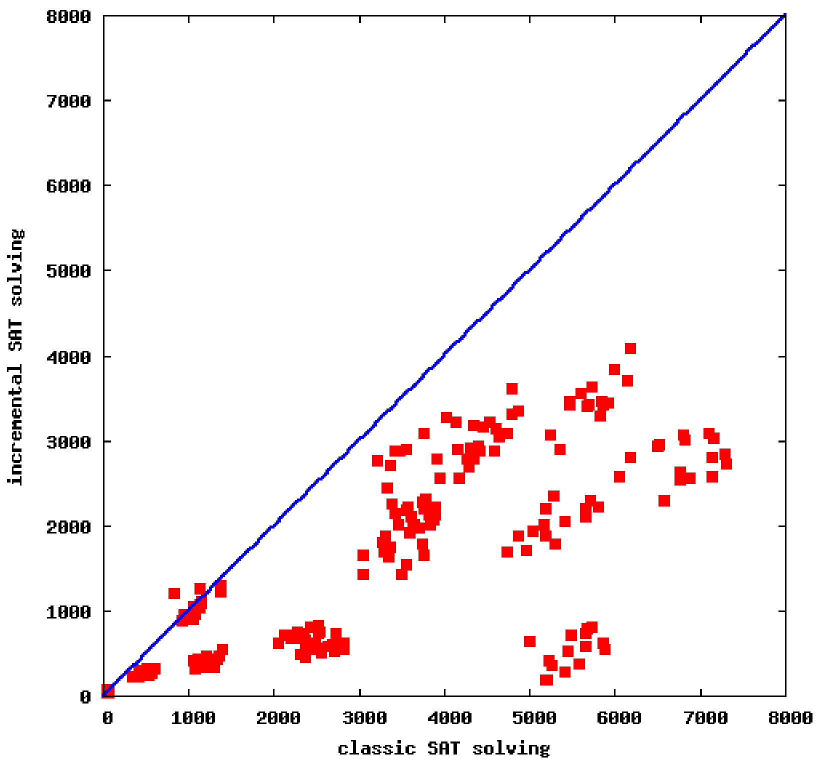


Figure A.8: “das9206” benchmark

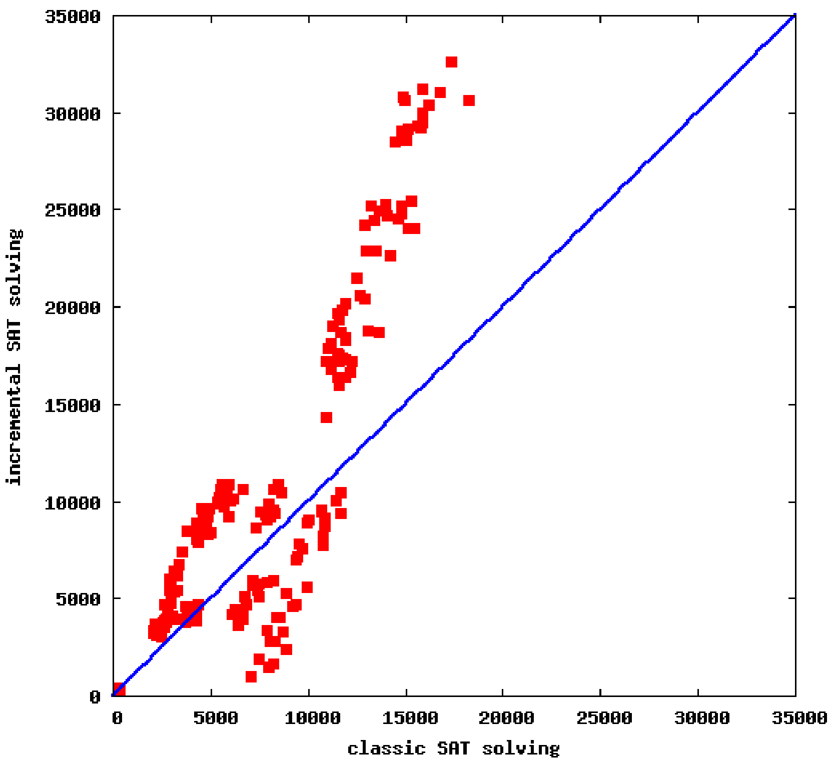


Figure A.9: “das9207” benchmark

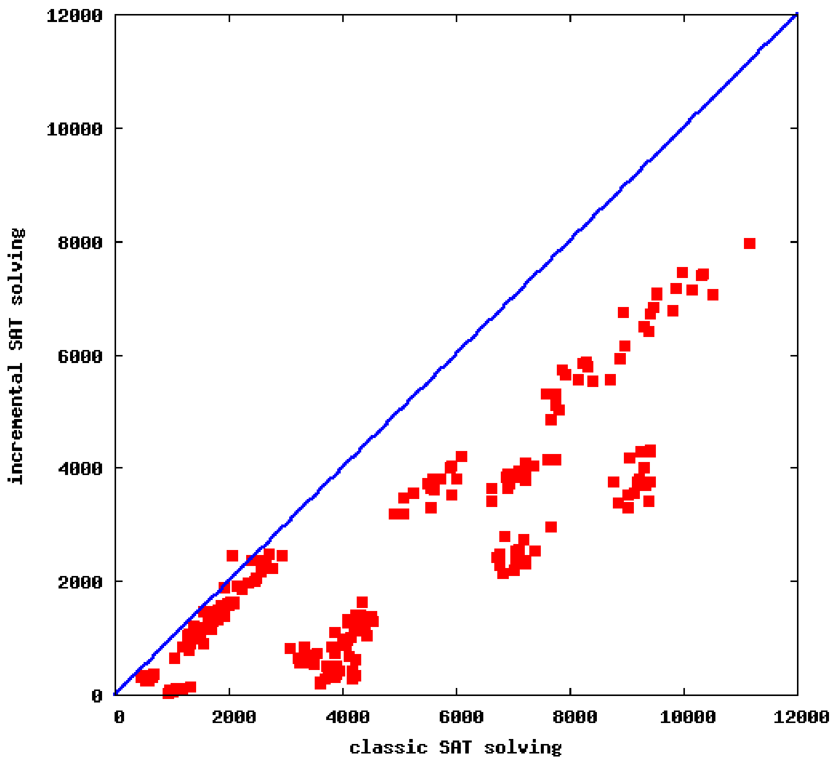


Figure A.10: “das9208” benchmark

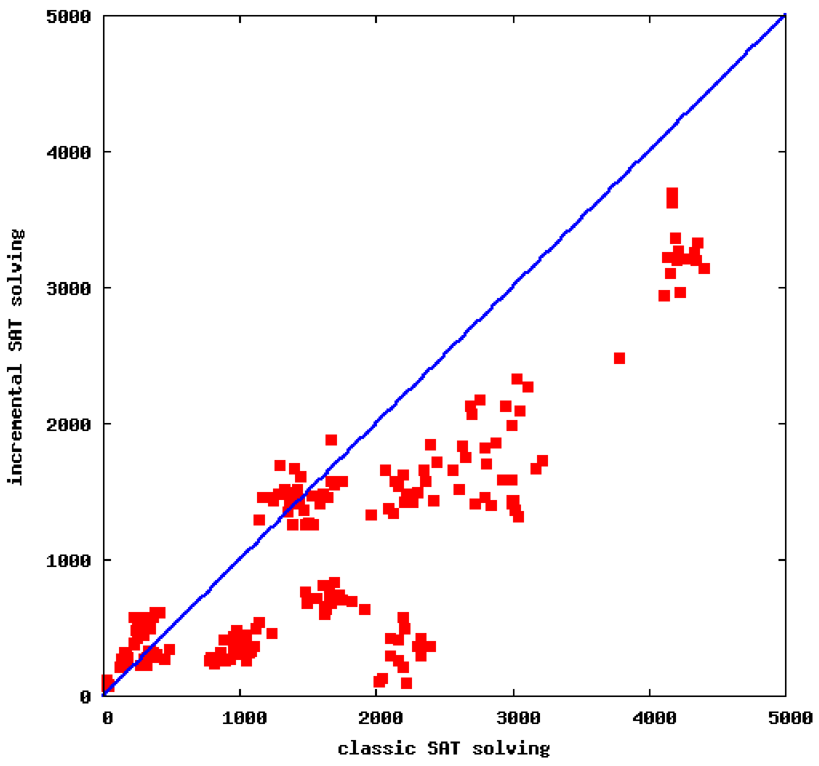


Figure A.11: “das9209” benchmark

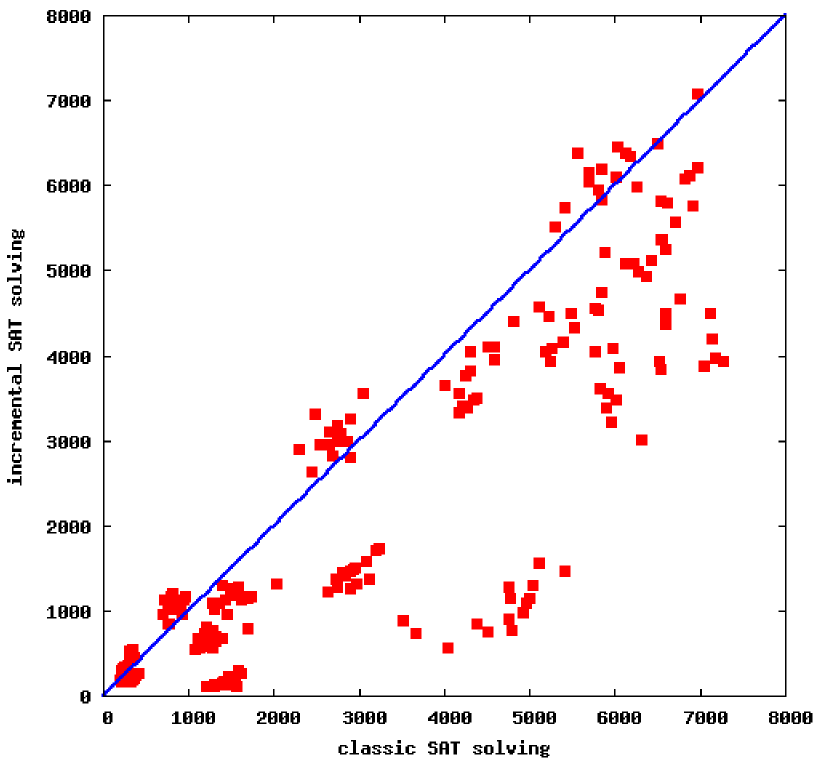


Figure A.12: “edf9201” benchmark

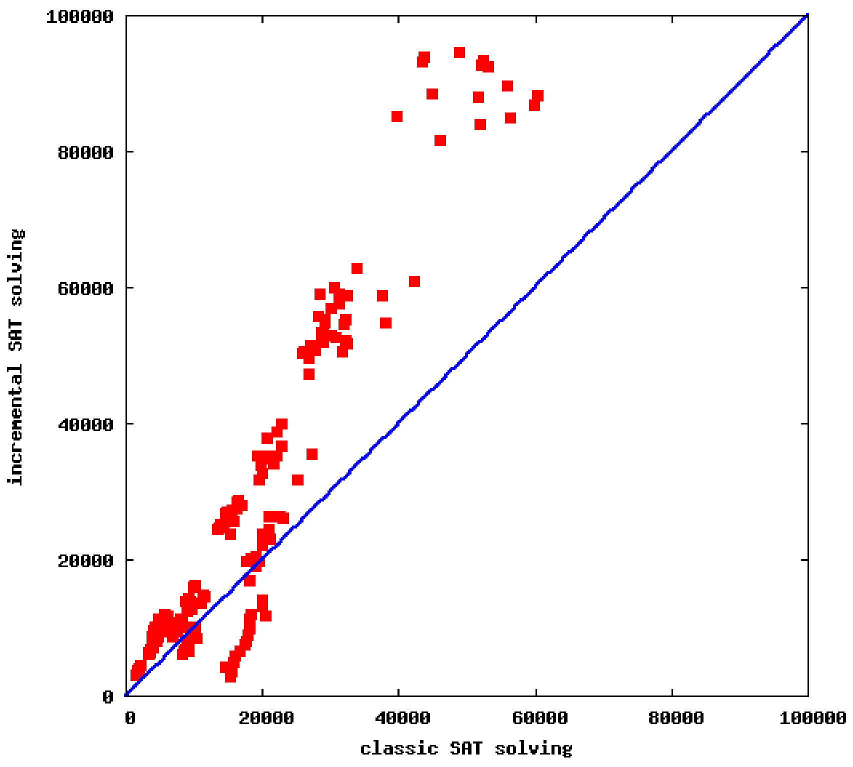


Figure A.13: “edf9202” benchmark

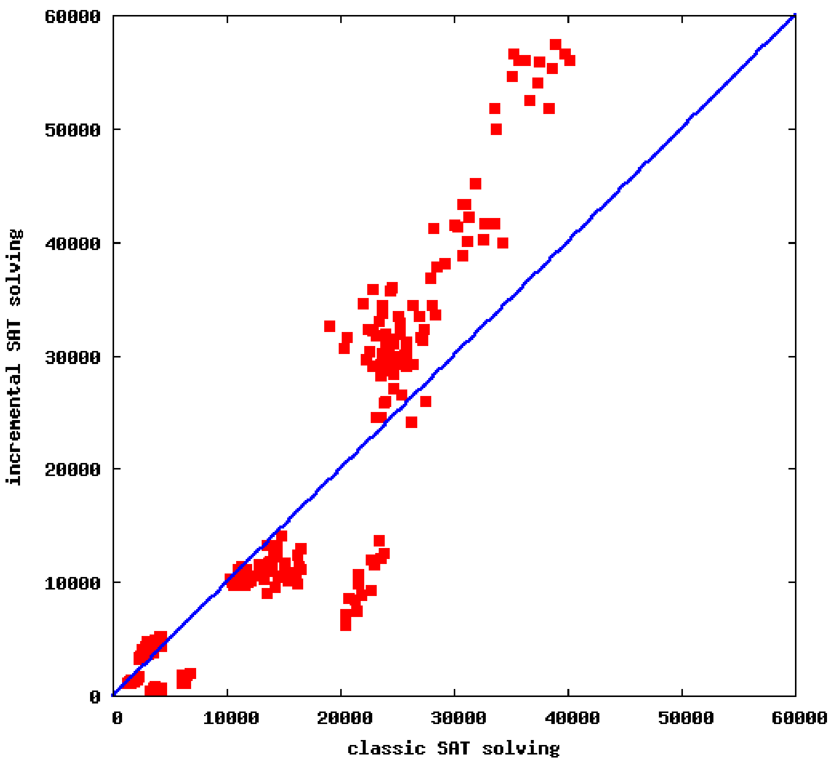


Figure A.14: “edf9203” benchmark

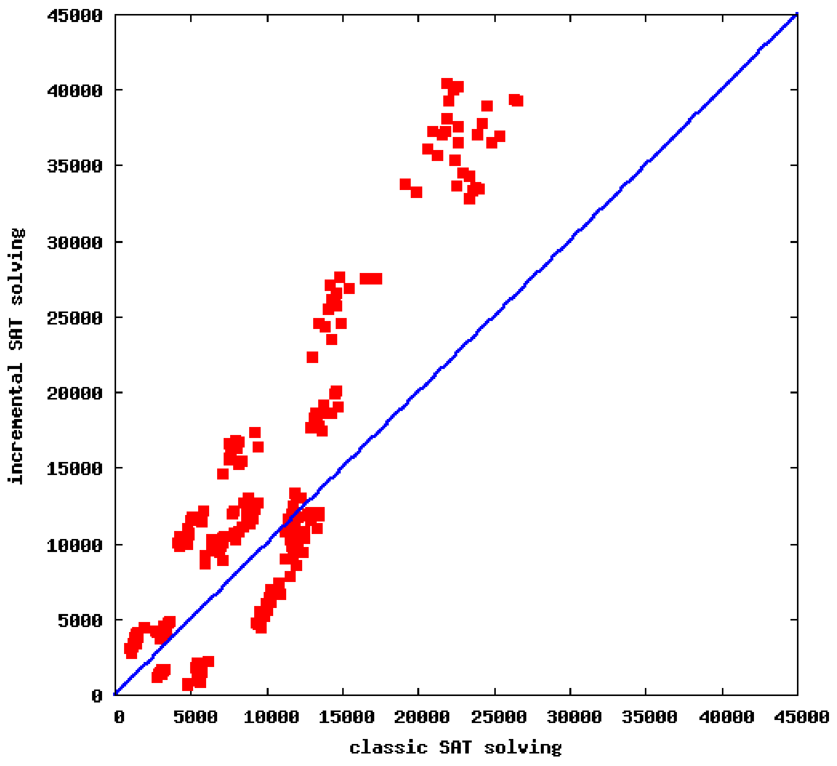


Figure A.15: “edf9204” benchmark

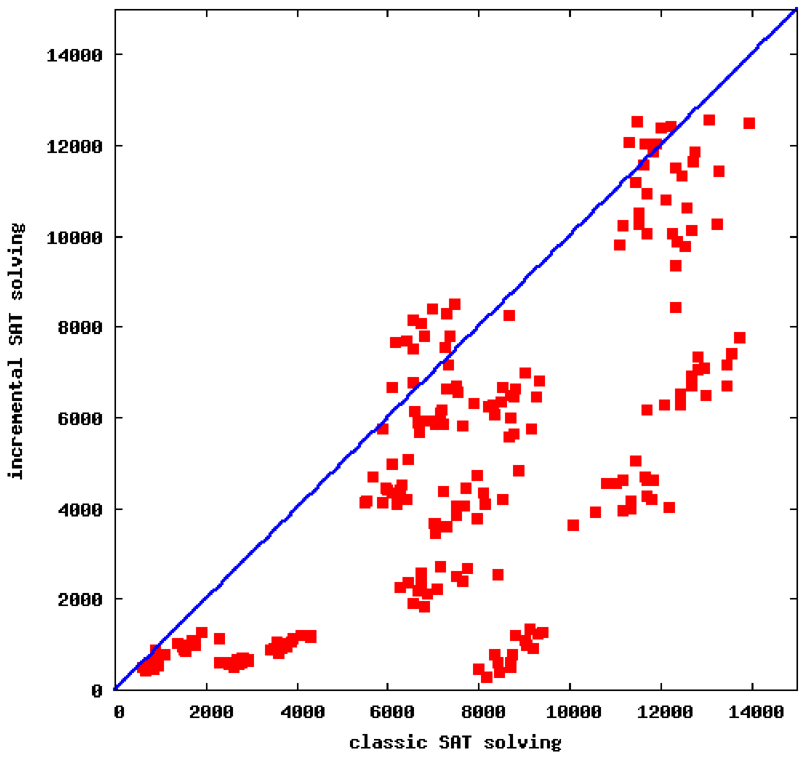


Figure A.16: "edf9205" benchmark

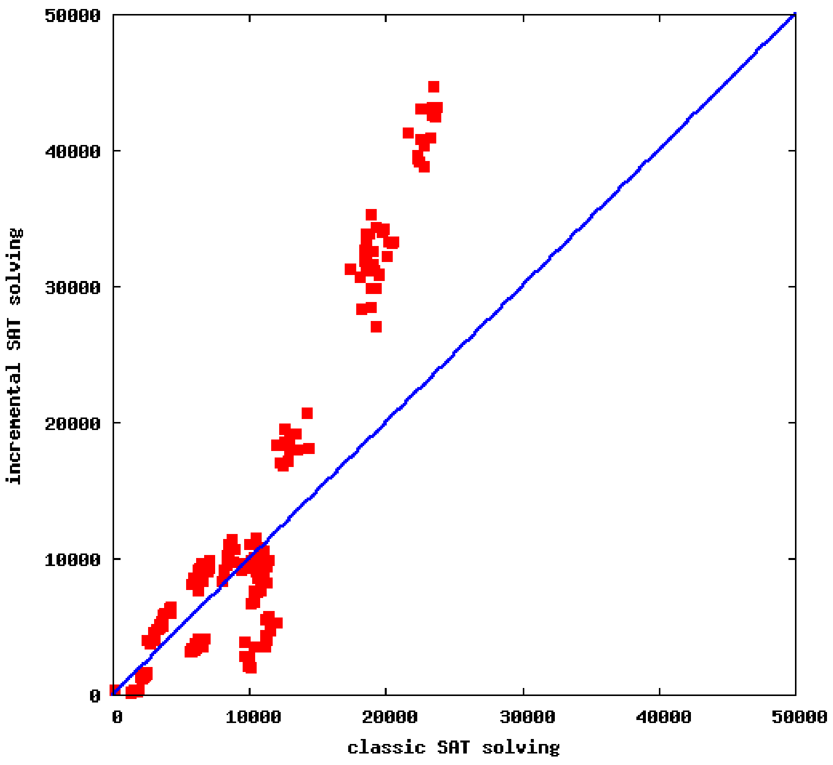


Figure A.17: “edf9206” benchmark

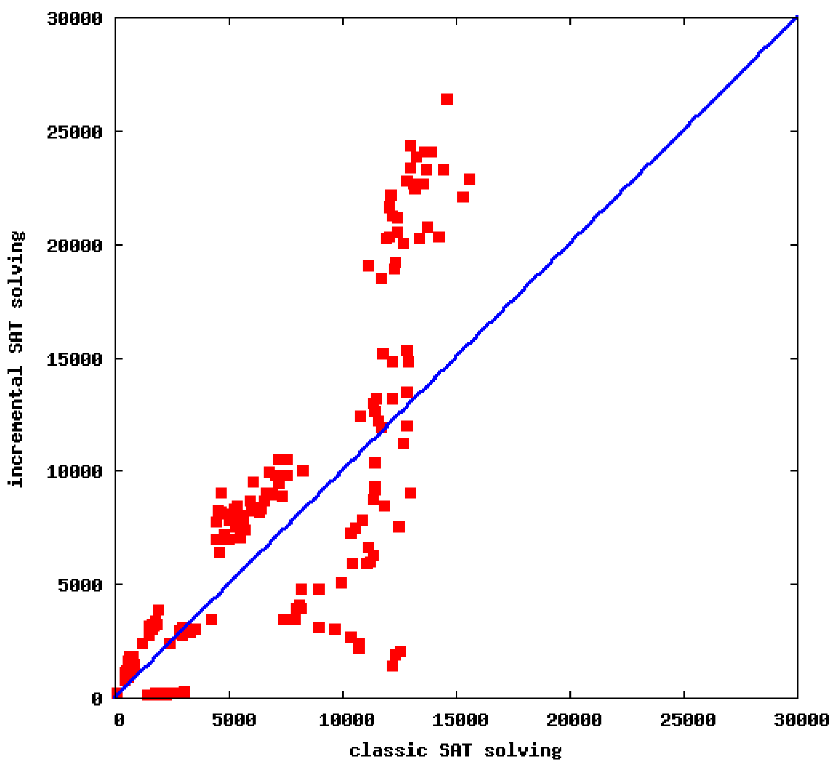


Figure A.18: “edfpa14b” benchmark

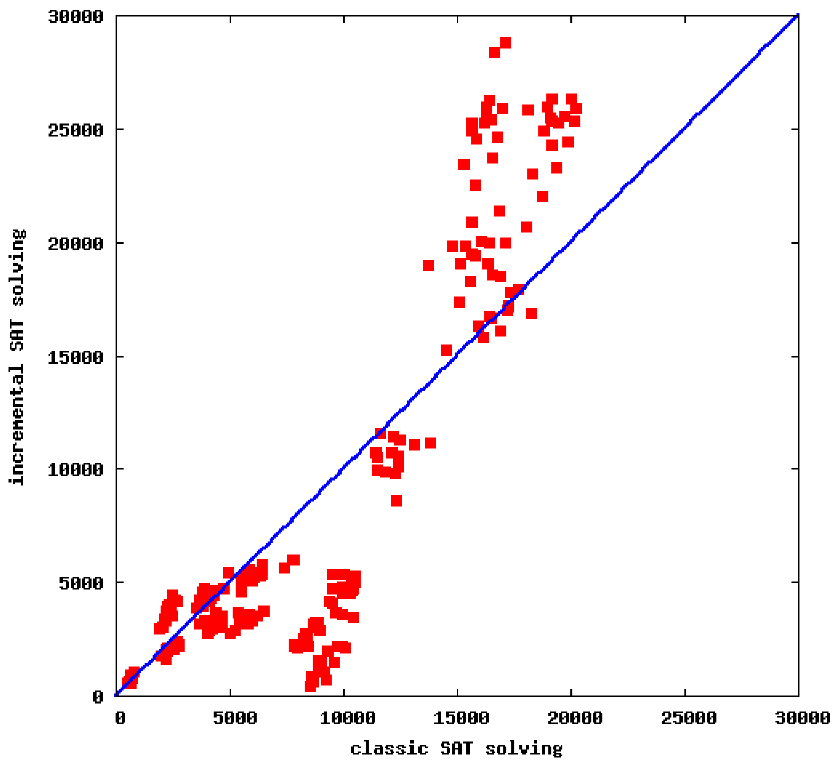


Figure A.19: “edfpa14o” benchmark

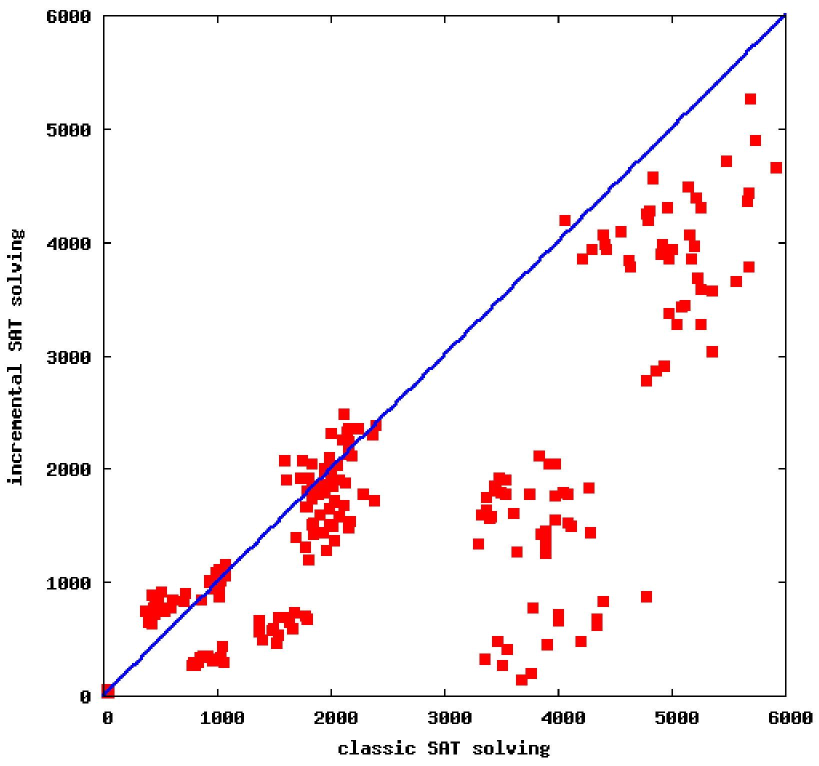


Figure A.20: “edfpa14p” benchmark

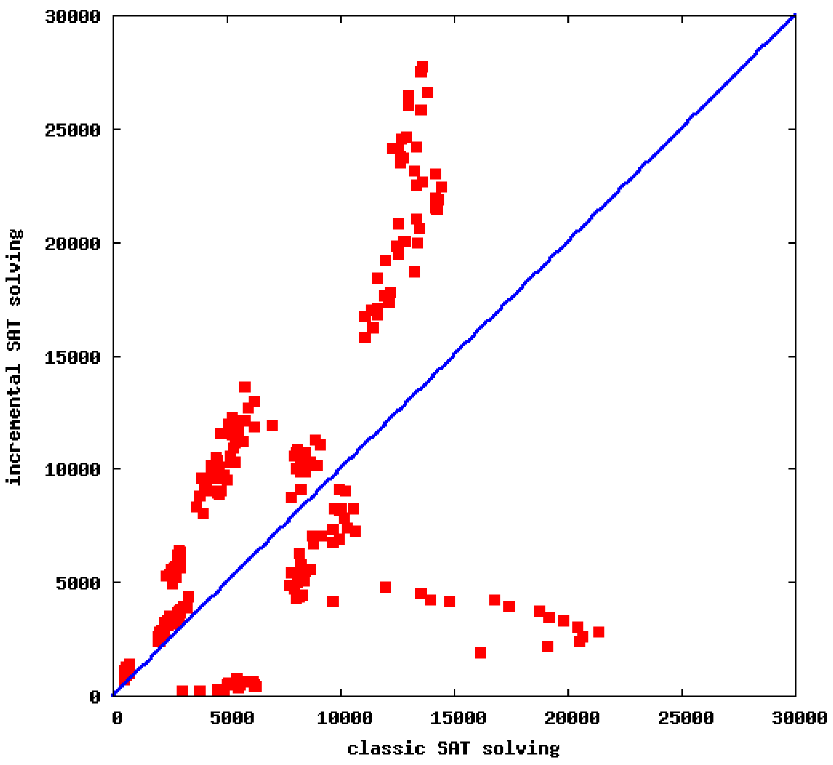


Figure A.21: “edfpa14q” benchmark

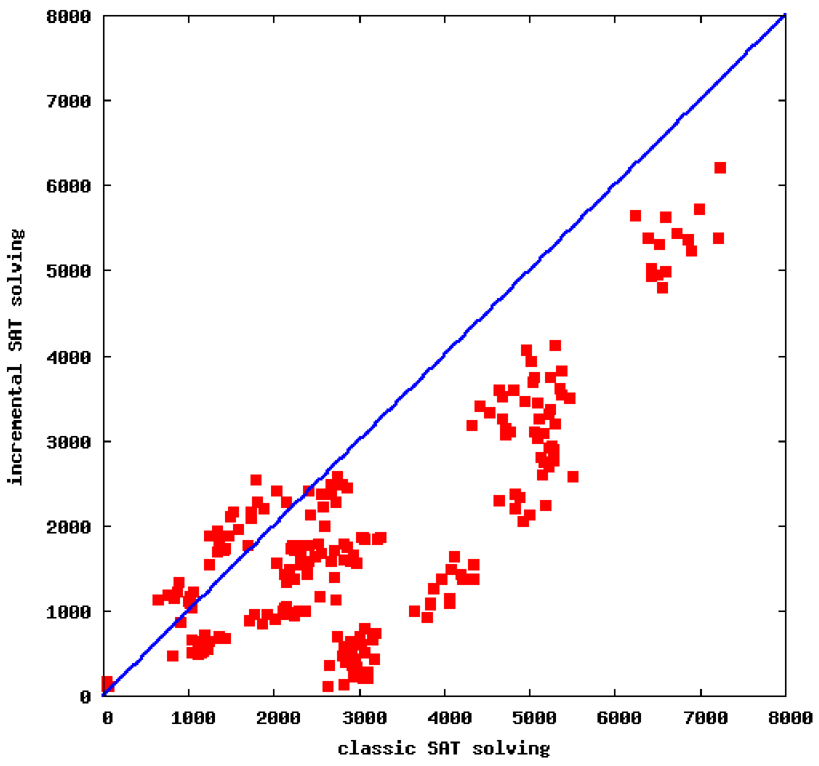


Figure A.22: “edfpa14r” benchmark

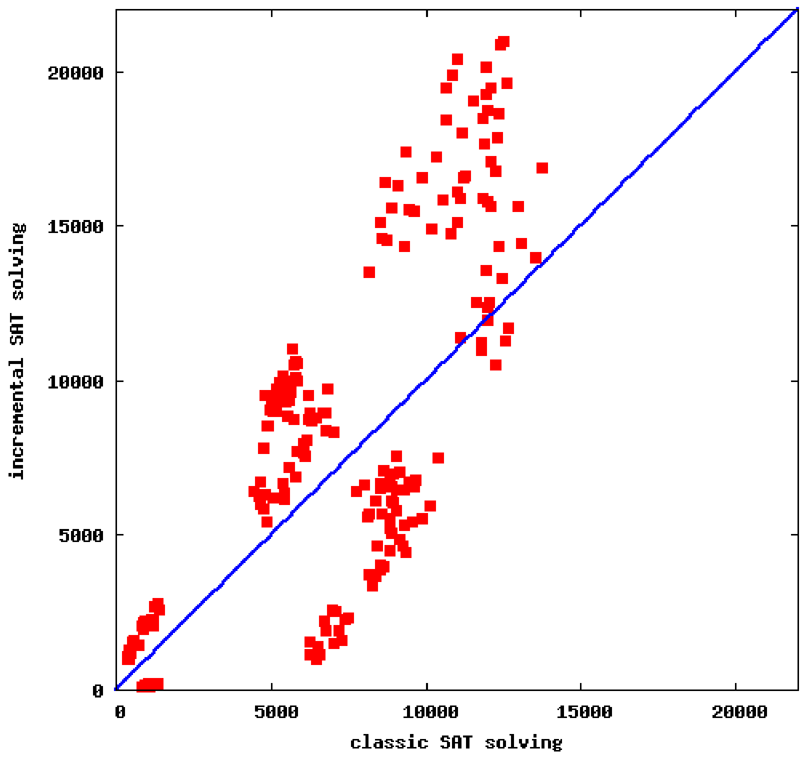


Figure A.23: “edfpa15b” benchmark

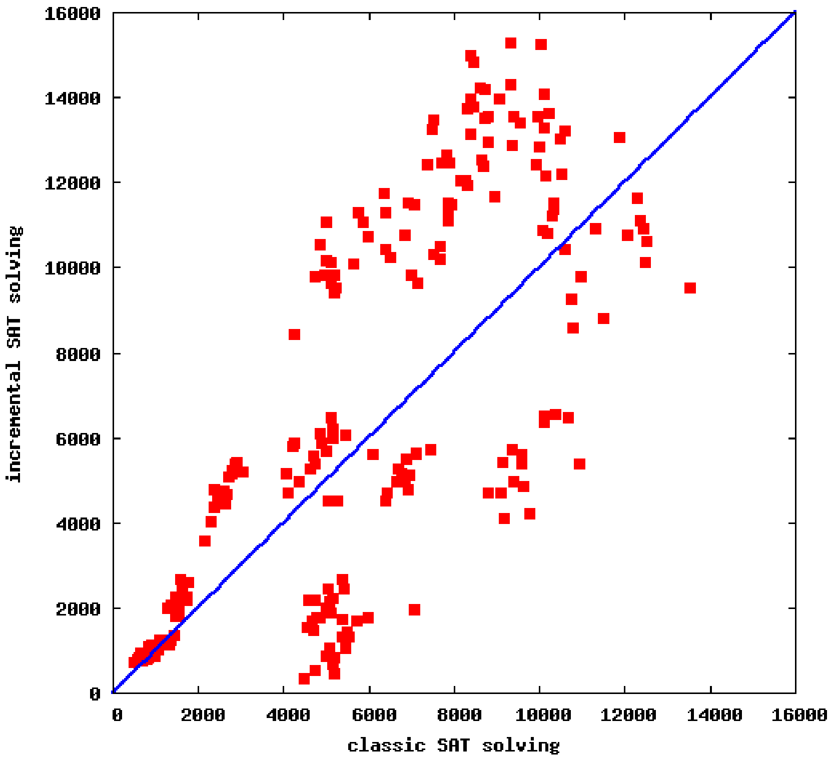


Figure A.24: “edfpa15o” benchmark

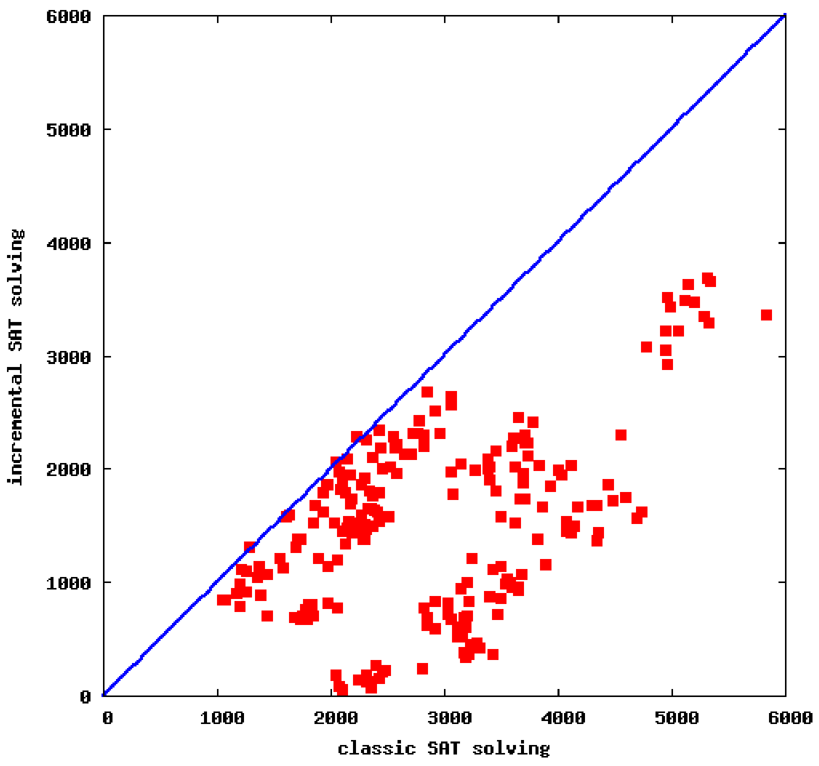


Figure A.25: “edfpa15p” benchmark

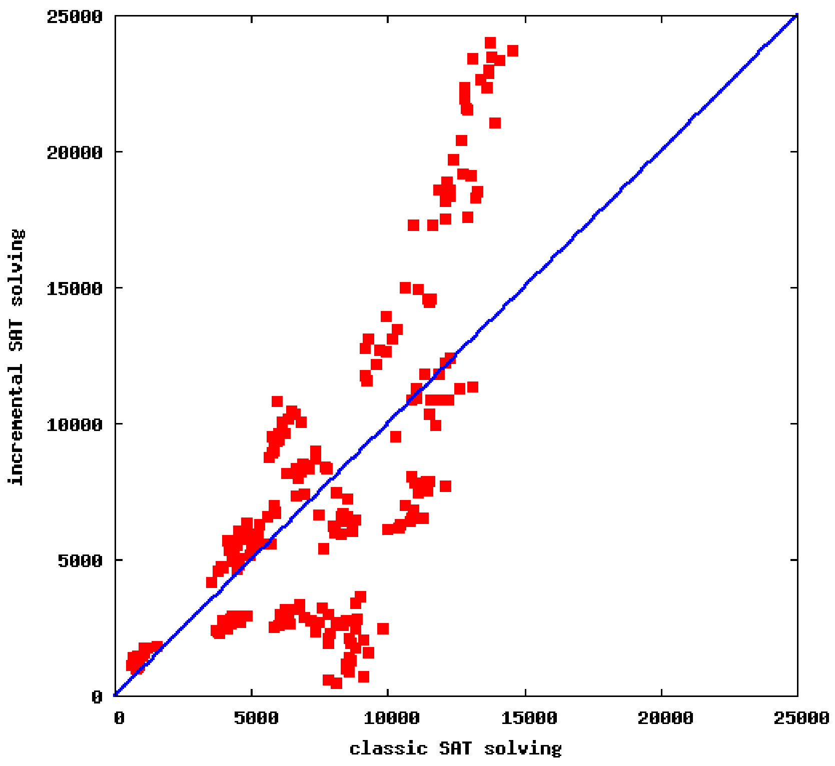


Figure A.26: “edfpa15q” benchmark

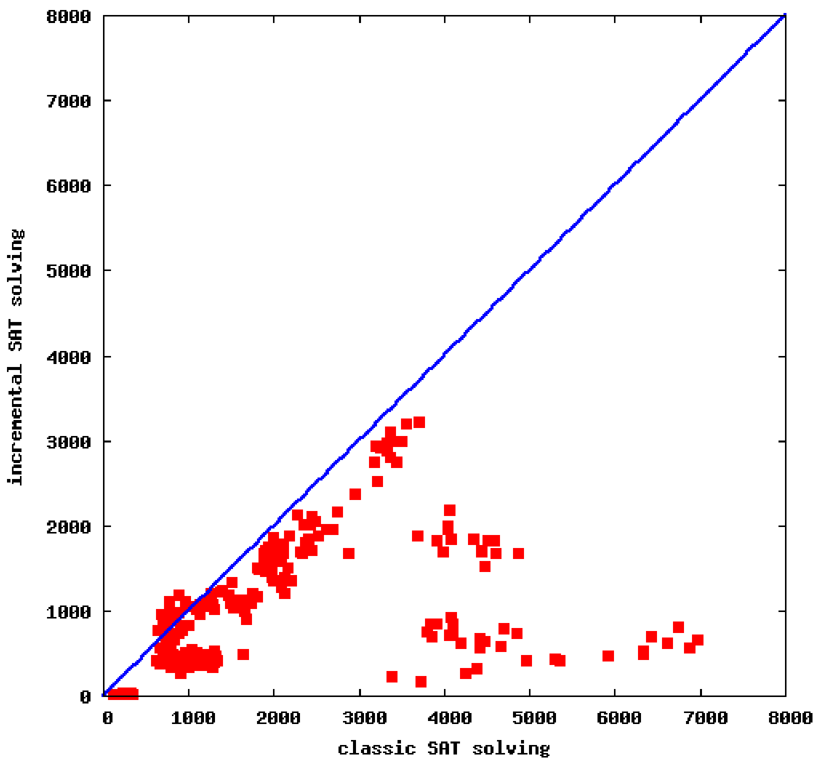


Figure A.27: “edfpa15r” benchmark

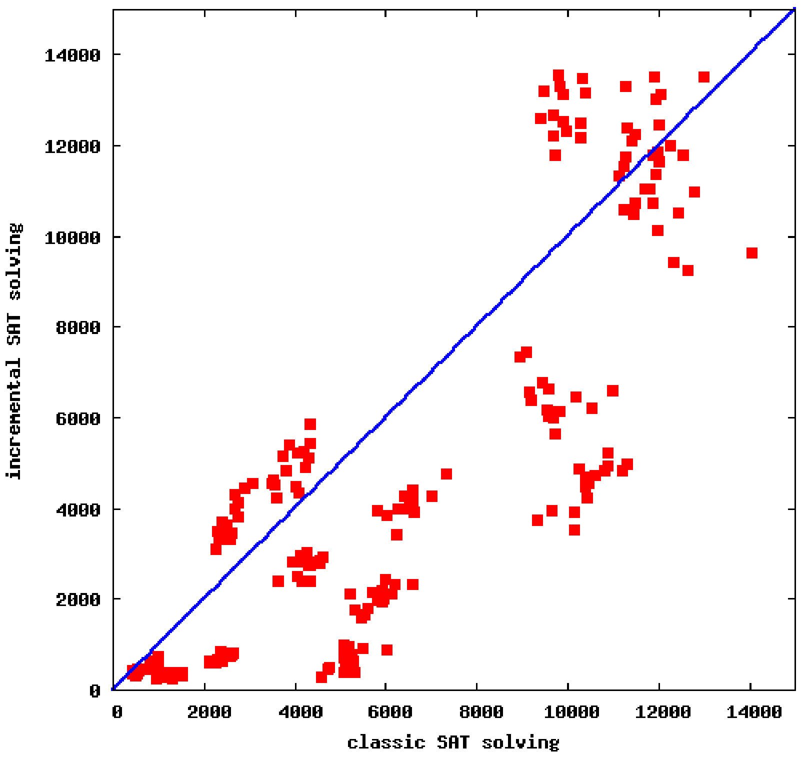


Figure A.28: “elf9601” benchmark

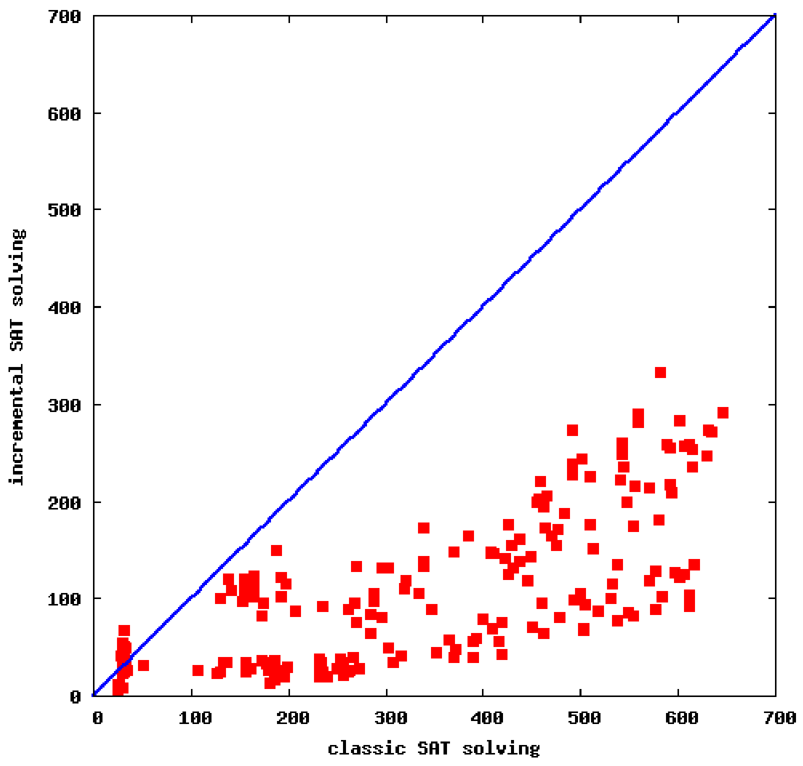


Figure A.29: “esvs” benchmark

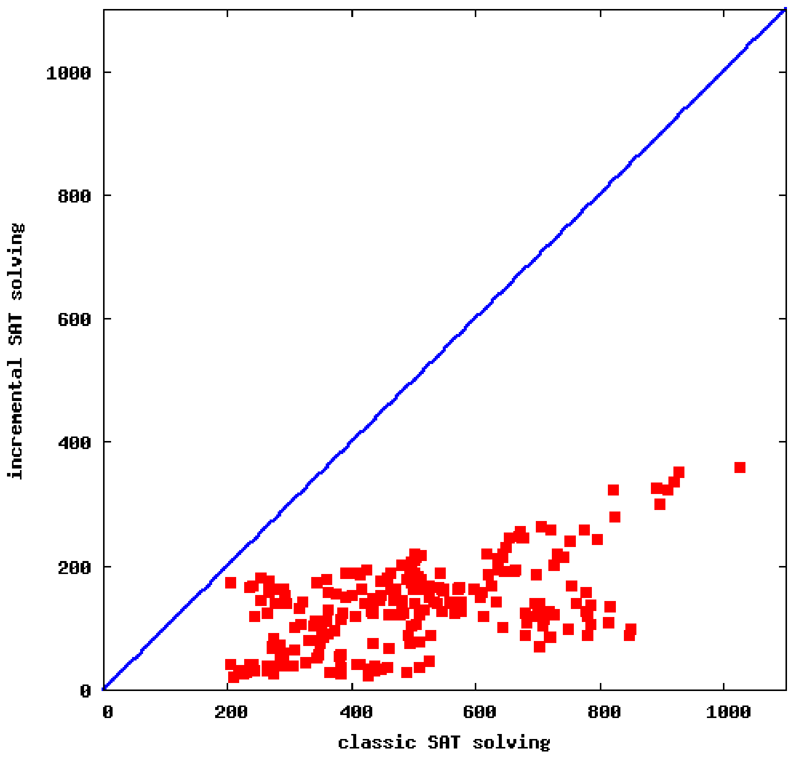


Figure A.30: “fs” benchmark

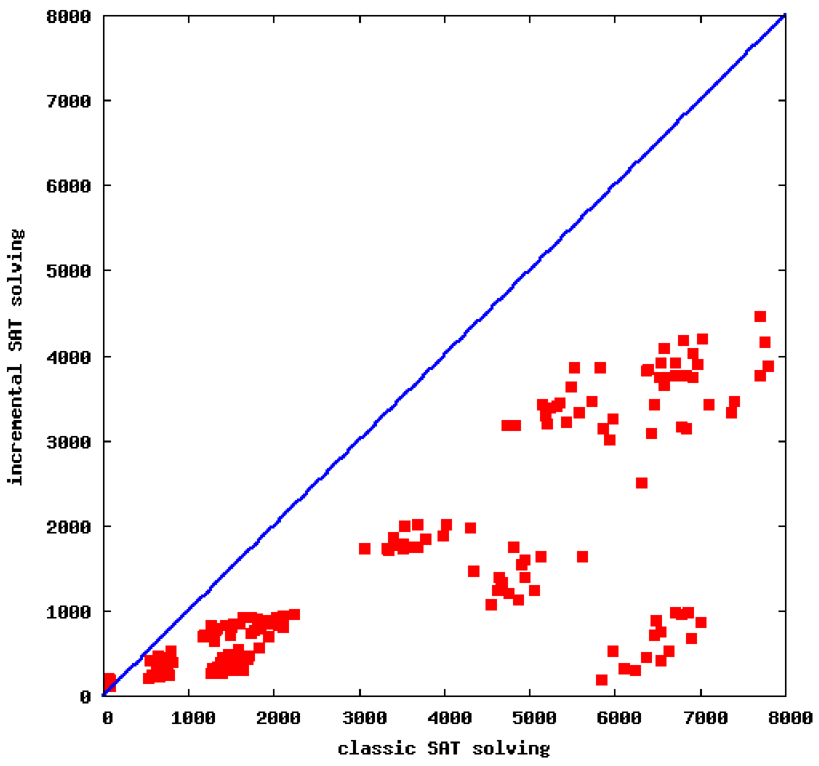


Figure A.31: “ftr10” benchmark

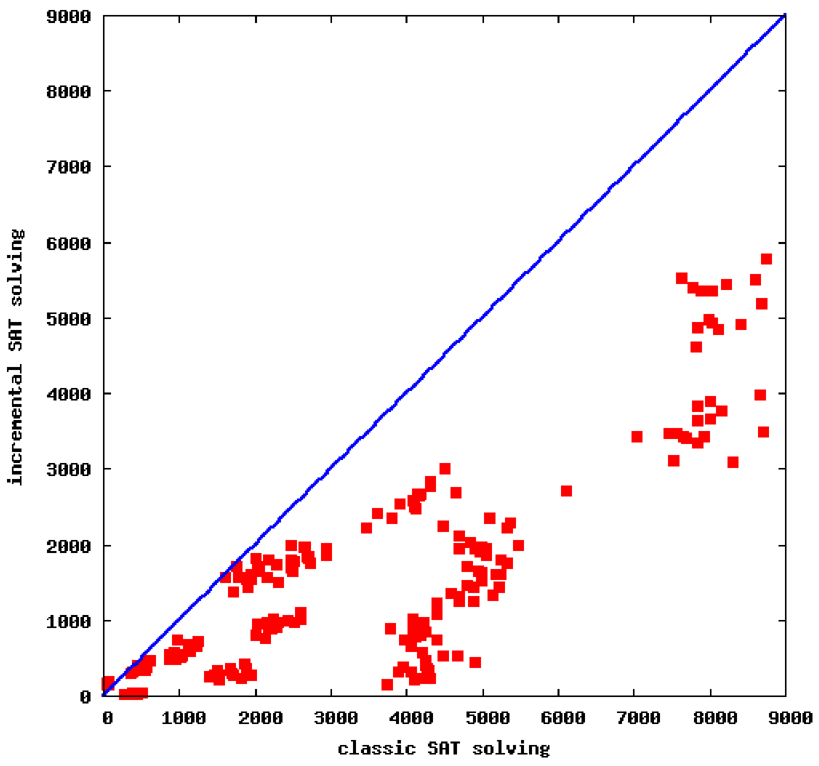


Figure A.32: “isp9602” benchmark

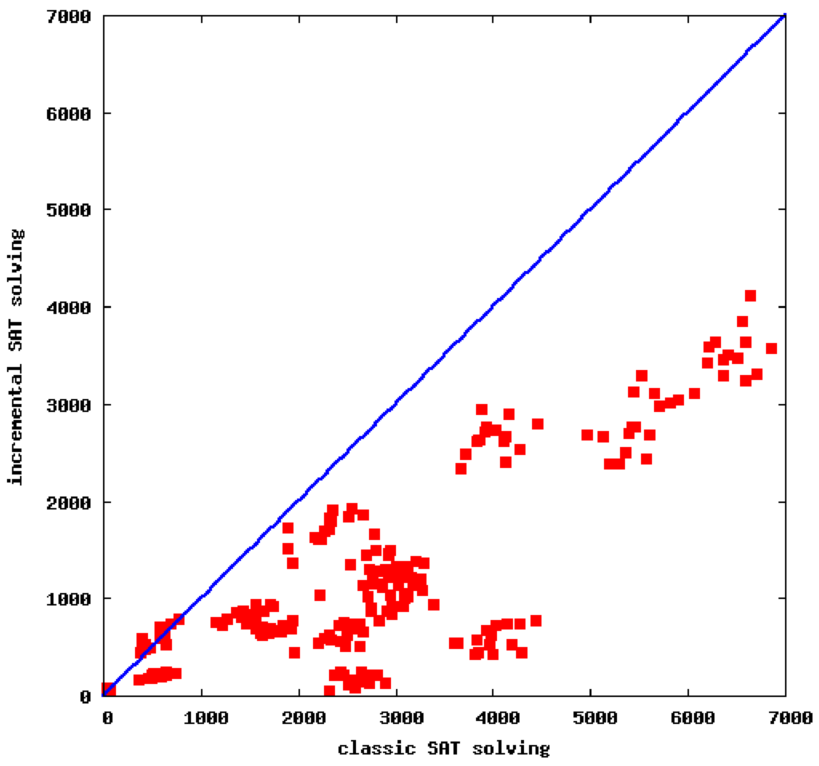


Figure A.33: “isp9603” benchmark

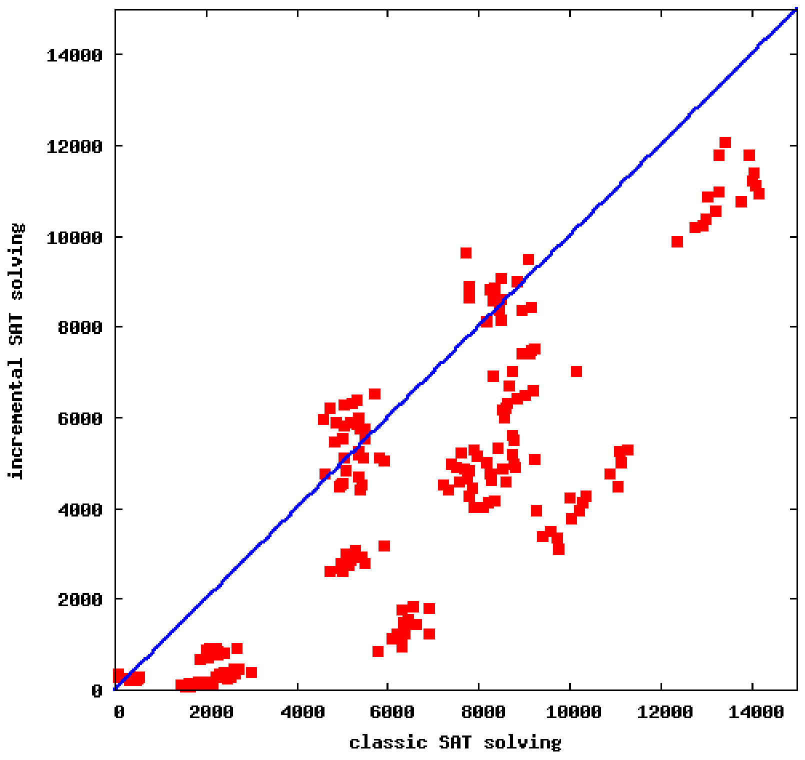


Figure A.34: “isp9604” benchmark

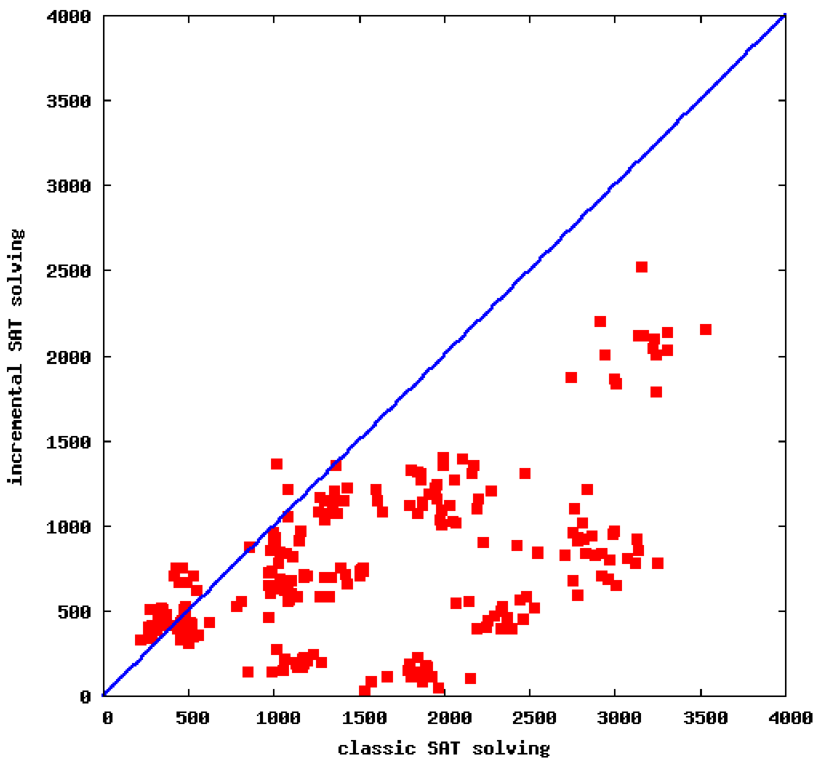


Figure A.35: “isp9606” benchmark

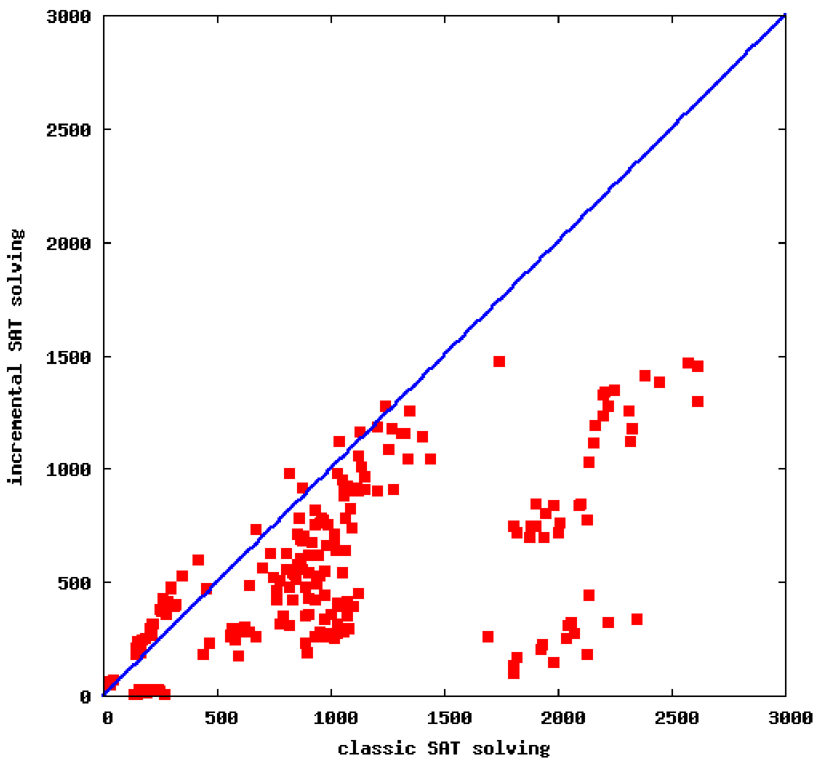


Figure A.36: “isp9607” benchmark

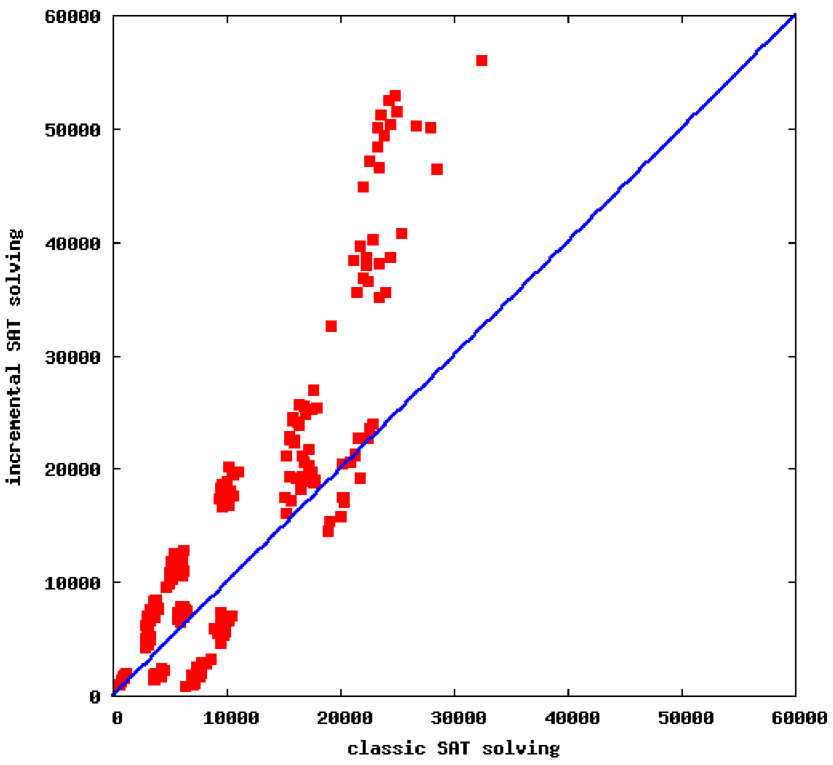


Figure A.37: “jbd9601” benchmark

Bibliography

[1] EASTWEB: Building an integrated leading Euro-Asian higher education and research community in the field of the SemanTic WEB http://www.eastweb.eu/

[2] Jilin University http://en.jlu.edu.cn/

[3] U. Junker. QuickXplain: Preferred Explanations and Relaxations for Over-Constrained Problems. *AAAI* (2004): 167-172. http://wikix.ilog.fr/wiki/pub/Main/UlrichJunker/pxpl.pdf

[4] U. Junker. QuickXplain: Preferred Explanations and Relaxations for Over-Constrained Problems (slides). http://wikix.ilog.fr/wiki/pub/Main/UlrichJunker/pxpl-talk.pdf

[5] U. Junker. QuickXplain: Conflict Detection for Arbitrary Constraint Propagation Algorithms. *IJCAI* (2001).

[6] J. Amilhastre, H. Fargier, P. Marquis. Consistency Restoration and Explanations in Dynamic CSPs - Application to Configuration, *Artificial Intelligence* 135(1-2): 199-234 (2002). ftp://ftp.irit.fr/pub/IRIT/RPDMP/AIJfinal1.pdf

[7] CLib http://www.itu.dk/research/cla/externals/clib/

[8] CLab http://www.itu.dk/people/rmj/data/systems/clab10/

[9] R. M. Jensen. CLab 1.0 User Manual, *Technical Report ITU-TR-2004-46, IT University of Copenhagen* (2004) http://www.itu.dk/people/rmj/data/systems/clab10/man.pdf

[10] MiniSat http://minisat.se/

[11] SAT Competition http://www.satcompetition.org/

[12] SAT-Race 2006 http://fmv.jku.at/sat-race-2006/

[13] N. Eén, N. Sörensson. An Extensible SAT Solver. Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing, SAT (2003)

[14] N. Eén, N. Sörensson. Temporal Induction by Incremental SAT Solving. Proceedings of the First International Workshop on Bounded Model Checking, 2003.

[15] R. Dechter. Constraint Processing, Morgan Kaufmann (2003).

[16] M. R. Garey, D. S. Johnson. Computers and Intractability: A Guide to the Theory of NP-completeness, Freeman (1979).

[17] S. A. Cook. The Complexity of Theorem-Proving Procedures. *Proceedings of the Third Annual ACM Symposium on Thoery of Computing* (1971): 151-158.

[18] S. Subbarayan, R. M. Jensen, T. Hadzic, H. R. Andersen, H. Hulgaard, J. Møller. Comparing Two Implementations of a Complete and Backtrack-Free Interactive Configurator. *Proceedings of Workshop on CSP Techniques with Immediate Application, CP04* (2004): 97-111. http://www.itu.dk/people/sathi/papers/cspia2004.pdf

[19] T. Hadzic, S. Subbarayan, R. M. Jensen, H. R. Andersen, J. Møller, H. Hulgaard. Fast Backtrack-Free Product Configuration Using a Precompiled Solution Space Representation. *Proceedings of the International Conference on Economic, Technical and Organisational Aspects of Product Configuration Systems* (2004): 131-138. http://www.itu.dk/people/sathi/papers/PETO2004.pdf

[20] S. Subbarayan, H. R. Andersen. Integrating a Variable Ordering Heuristic with BDDs and CSP Decomposition Techniques for Interactive Configurators (2005) http://www.itu.dk/people/sathi/papers/tob-var.pdf

[21] S. Subbarayan. Integrating CSP Decomposition Techniques and BDDs for Compiling Configuration Problems. *Proceedings of the International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, CPAIOR* (2005): 351-365 http://www.itu.dk/people/sathi/papers/icode.pdf

[22] R. Sebastiani. Lazy Satisfiability Modulo Theories. *Journal on Satisfiability, Boolean Modeling and Computation* 3 (2007): 141-224. http://jsat.ewi.tudelft.nl/content/volume3/JSAT3_9_Sebastiani.pdf

[23] A. Cimatti, R. Sebastiani. Building Efficient Decision Procedures on Top of SAT Solvers. *6th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Hardware Verification* (2006). Volume 3965 of *LNCS*, Springer Verlag.

[24] E. Goldberg, Y. Novikov. BerkMin: A Fast and Robust SAT-Solver. *Proceedings DATE '02*, page 142, Washington, DC, USA, 2002. IEEE Computer Society.

[25] R. Brafman. A Simplifier for Propositional Formulas with many Binary Clauses. *Proceedings IJCAI01*, 2001.

[26] R. J. Bayardo, R. C. Schrag. Using CSP Look-Back Techniques to Solve Real-World SAT instances. *Proceedings AAAI'97*, pages 203-208. AAAI Press, 1997.

[27] F. Bacchus, J. Winter. Effective Preprocessing with Hyper-Resolution and Equality Reduction. *Proceedings Sixth International Symposium on Theory and Applications of Satisfiability Testing*, 2003.

[28] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Introduction to Algorithms*. MIT Press, 1998.

[29] M. Davis, G. Logemann, D. Loveland. A Machine Program for Theorem Proving. *Journal of the ACM*, 5(7), 1962.

[30] M. Davis, H. Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7:201-215, 1960.

[31] N. Eén, A. Biere. Effective Preprocessing in SAT Through Variable and Clause Elimination. *Proceedings SAT'05*, volume 3569 of *LNCS*. Springer, 2005.

[32] E. Giunchiglia, A. Massarotto, R. Sebastiani. Act, and the Rest Will Follow: Exploiting Determinism in Planning as Satisfiability. *Proceedings AAAI'98*, pages 948-953, 1998.

[33] C. P. Gomes, B. Selman, H. Kautz. Boosting Combinatorial Search Through Randomization. *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI'98)*, pages 431-437, Madison, Wisconsin, 1998.

[34] John N. Hooker, V. Vinay. Branching Rules for Satisfiability. *Journal of Automated Reasoning*, 15(3):359-383, 1995.

[35] R. G. Jeroslow, J. Wang. Solving Propositional Satisfiability Problems. *Annals of Mathematics and Artificial Intelligence*, 1(1-4):167-187, 1990.

[36] C. M. Li, Anbulagan. Heuristics Based on Unit Propagation for Satisfiability Problems. *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 366-371, 1997.

[37] C. M. Li, Anbulagan. Look-Ahead Versus Look-Back for Satisfiability Problems. *Principles and Practice of Constraint Programming*, volume 1330 of *LNCS*, 1997.

[38] C. M. Li. Integrating Equivalency Reasoning into Davis-Putnam Procedure. *AAAI: 17th National Conference on Artificial Intelligence.* AAAI / MIT Press, 2000.

[39] M. W. Moskewicz, C. F. Madigan, Y. Z., L. Zhang, S. Malik. Chaff: Engineering an Efficient SAT solver. *Design Automation Conference*, 2001.

[40] R. Nieuwenhuis, A. Oliveras. DPLL(T) with Exhaustive Theory Propagation and its Application to Difference Logic. *Proceedings CAV'05*, volume 3576 of *LNCS*. Springer, 2005.

[41] R. Nieuwenhuis, A. Oliveras, C. Tinelli. Abstract DPLL and Abstract DPLL Modulo Theories. *Proceedings LPAR'04*, volume 3452 of *LNCS*. Springer, 2005.

[42] R. Nieuwenhuis, A. Oliveras, C. Tinelli. Solving SAT and SAT Modulo Theories: from an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). *Journal of the ACM*, 53(6):937-977, November 2006.

[43] J. P. M. Silva, K. A. Sakallah. GRASP - A New Search Algorithm for Satisfiability. *Proceedings ICCAD'96*, 1996.

[44] O. Strichman. Tuning SAT Checkers for Bounded Model Checking. *Proceedings CAV00*, volume 1855 of *LNCS*, pages 480-494. Springer, 2000.

[45] C. Tinelli. A DPLL-based Calculus for Ground Satisfiability Modulo Theories. *Proceedings JELIA-02*, volume 2424 of *LNAI*, pages 308-319. Springer, 2002.

[46] L. Zhang, S. Malik. The Quest for Efficient Boolean Satisfiability Solvers. *Proceedings CAV'02*, number 2404 in *LNCS*, pages 17-36. Springer, 2002.

[47] L. Zhang, C. F. Madigan, M. W. Moskewicz, S. Malik. Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. *ICCAD*, pages 279-285, 2001.

[48] Ubuntu http://www.ubuntu.com/

[49] Debian http://www.debian.org/

[50] Java http://java.sun.com/

[51] NetBeans IDE http://www.netbeans.org/

[52] ANTLR: ANother Tool for Language Recognition http://www.antlr.org/

[53] JGraphT http://jgrapht.sourceforge.net/

[54] ILOG http://www.ilog.com/

[55] Configit A/S http://www.configit.com/

[56] Choco http://choco-solver.net/

[57] Comet http://www.comet-online.org/

[58] JaCoP http://jacop.cs.lth.se/

[59] Cream http://bach.istc.kobe-u.ac.jp/cream/

[60] Koalog http://www.koalog.com/

[61] Gecode http://www.gecode.org/

Acknowledgments

I acknowledge Prof. Roberto Sebastiani for his accurate supervision of my whole work, the implementation of the tool and the writing of the thesis. Many thanks also go to Prof. Zhanshan Li, Yanggong Zhang, Wencheng Han and Michele Vescovi that helped me several times in the understanding of some theoretical parts and in the design of the tool. Moreover I want to remember my parents because they always supported me for my studies.