# SAT with Global Constraints

Md Solimul Chowdhury
Department of Computing Science,
University of Alberta,
Edmonton, Alberta, Canada
Email: mdsolimu@ualberta.ca

Jia-Huai You
Department of Computing Science,
University of Alberta,
Edmonton, Alberta, Canada
Email: you@cs.ualberta.ca

***Abstract*—We present a tight integration of SAT with CP, called SAT(gc), which embeds global constraints into SAT. A prototype is implemented by integrating the state of the art SAT solver ZCHAFF and the generic constraint solver GECODE. Experiments are carried out for benchmarks from puzzle domains and planning domains to reveal insights in compact representation, solving effectiveness, and novel usability of the new framework.**

## I. Introduction

SAT solving is to encode a given problem by a collection of propositional clauses and use a SAT solver to determine whether there exists a variable assignment that satisfies every clause. Modern SAT solvers are fairly efficient, largely due to conflict-directed learning and backtracking, and have been applied to a number of problem domains, such as AI planning, bounded model checking, software diagnosis, theorem proving for subsets of first-order logic [1].

Constraint Programming (CP), on the other hand, has been applied to solving scheduling, verification, and some other combinatorial search problems [2]. Historically, CP is developed from the Constraint Satisfaction Problem (CSP). For practical applications, languages (such as GECODE[1], Choco[2], ECLiPSe[3]) have been developed to facilitate the definitions of constraints in terms of primitive and built-in constraints. One kind of built-in constraints are called *global constraints*, which are pre-defined constraints over a non-fixed number of variables [3]. The use of global constraints not only facilitates problem representation, but also enables efficient reasoning based on special data structures and dedicated constraint propagators.

However, SAT and CP have their own weaknesses. For example, SAT is incapable of compactly representing numerical constraints, and due to the flat structure of clauses, useful structured information of a given domain tends to get lost in a clausal representation. On the other hand, CP is primarily developed for solving constraints over variables with non-trivial domains. Some real world problems do not fall into this category, for which CP solving seems not as effective as SAT solving. AI planning and bounded model checking show characteristics of problems of this kind. Furthermore, CP deals with a collection of constraints, not their possible combinations. As a result, compositions of constraints, such as conditional constraints (called *dynamic constraints*) and disjunctive constraints, often present additional challenges and require special treatments [4], [5].

In recent years, cross fertilization of these two areas has become a topic of interest. It is argued that complex real world applications may require effective features of both [1]. A number of approaches have been pursued in this direction recently. For example, in SAT Modulo Theory (SMT) [6], theory solvers of various kinds are incorporated into a SAT solver, where part of the problem is encoded in an embedded theory and solved by a dedicated theory solver. To deal with numeric constraints, the SAT community has moved to a different direction - *pseudo Boolean constraints*, where constraints are expressed by linear inequalities over sum of weighted Boolean functions (see, e.g., [7]). In [8], a framework for integrating CSP style constraint solving in Answer Set Programming (ASP) has been developed. More recently, the usefulness of combining ASP and CP for industrial sized problems is demonstrated in [9].

In this paper we develop a tight integration of CSP into SAT, called SAT$(gc)$. We focus on embedding global constraints into SAT. Since any user defined constraint in CP can be treated as a "global constraint", this integration essentially becomes one of integrating CP into SAT, with the feature that when one is absent, the system behaves like the other. In this way, unlike pseudo Boolean constaints, where SAT is extended as a special case, this integration can be seen as a proper extension of SAT as well as CP.

The rest of this paper is organized as follows. In the next section, we briefly discuss preliminaries of SAT. In Section III, we define the language of SAT$(gc)$ and provide some notations and examples. Then in Section IV, we develop a solver for SAT$(gc)$. We implemented a prototype called SATCP, which is discussed in Section V. Section VI describes the benchmarks used in our experiments with SATCP, their SAT$(gc)$ encoding, experimental results, and an analysis of these results. Related work is discussed in Section VII, with future directions discussed in Section VIII.

## II. SAT Preliminaries

In SAT, a formula is a finite set of clauses in propositional logic, where a clause is a disjunction of literals and a literal is either a proposition or its negation.

[1] http://www.gecode.org/

[2] www.emn.fr/z-info/choco-solver

[3] www.eclipseclp.org/

Let $V$ be a finite set of propositional symbols, called $variables$. For any $v \in V$, $v$ and $\neg v$ are called the *literals* of $v$ denoting the positive and negative phases of variable $v$, respectively. A $clause$ is a disjunction of literals $l_1 \vee \cdots \vee l_n$. A SAT $formula$ is a conjunction of one or more clauses $c_1 \wedge \cdots \wedge c_n$ [6].

Given a propositional formula in the clausal form, the task of determining whether there exists a variable assignment such that the formula evaluates to true is called the Boolean Satisfiability Problem, abbreviated as SAT [10].

## III. Language and Notation

In the language of SAT, propositions are also called *variables*. To distinguish, let us call these variables *normal variables*. In the language of $\mathrm{SAT}(gc)$, we have two additional types of variables/literals. The first is called a *global constraint literal*, or just a *gc-literal*, which represents a call to a global constraint. E.g., we can write a clause

$$allDiff(x_0 : \{v_1, v_2\}, x_1 : \{v_2, v_3\}) \vee \neg p$$

where the first disjunct is a call to the global constraint allDifferent in which $x_0$ and $x_1$ are CSP variables each of which is followed by its domain. In the sequel, we will use a named variable in the place of a gc-variable, with the correspondence between it and the (call to the) global constraint as part of a $\mathrm{SAT}(gc)$ instance.

A gc-literal is true if and only if the corresponding global constraint is solvable, which means that there exists one or more solutions for that gc-literal. Such a solution can be represented by a conjunction of propositional variables, each of which is a proposition representing that a given CSP variable takes a particular value from its domain. These new type of variables are called *value variables*. For each CSP variable $x$ and each value $a$ in its domain, we write $x_a$ for the corresponding value variable. Semantically, $x_a$ is true iff $x$ is assigned with value $a$. Since a value variable is just a proposition, it can appear in clauses of a $\mathrm{SAT}(gc)$ instance.

As a CSP variable cannot be assigned to more than one value from its domain, we impose the *exclusive value axioms (EVAs)*: for each CSP variable $x$ and distinct domain values $a$ and $b$, we have a clause $\neg x_a \vee \neg x_b$. In the sequel, we assume that EVAs are part of a $\mathrm{SAT}(gc)$ instance, so that *unit propagation* enforces these axioms automatically.

With the language of $\mathrm{SAT}(gc)$ defined above, given a $\mathrm{SAT}(gc)$ instance, a gc-variable in it is semantically equivalent to a disjunction of conjunctions of value variables, augmented by the EVAs, with each conjunction representing a solution of the corresponding global constraint (if such a disjunction is empty, it represents *false*). That is, a $\mathrm{SAT}(gc)$ instance is semantically equivalent to a propositional formula. Given a $\mathrm{SAT}(gc)$ instance $\Pi$, let us denote by $\sigma(\Pi)$ this propositional formula. We now can state precisely what the satisfiability problem in the current context is:

*Given a formula $\Pi$ in the language of $\mathrm{SAT}(gc)$, determine whether there exists a variable assignment such that $\sigma(\Pi)$ evaluates to true.*

### A. Representation power of $\mathrm{SAT}(gc)$

Let us consider some examples. In the first, suppose given a 4 by 4 board where each cell contains a number from a given domain $D$. We can express a disjunctive constraint, "at least one row has the sum of its numbers equal to a given number, say $k$", as follows

$$sum(x_{11}{:}D, \ldots, x_{14}{:}D,{=},k) \vee ... \vee sum(x_{41}{:}D \ldots, x_{44}{:}D,{=},k)$$

In $\mathrm{SAT}(gc)$ this can be written by a clause of four gc-variables: $v_{g_1} \vee v_{g_2} \vee v_{g_3} \vee v_{g_4}$, with the correspondence between the gc-variables and global constraints recorded as part of input instance. If, in addition, we want to express that there is exactly one of sum constraints that holds, we can write

$$\neg v_{g_i} \vee \neg v_{g_j} \qquad 1 \leq i, j \leq 4, i \neq j$$

As another example, suppose we want to represent a conditional constraint: given a graph and four colors, $\{r, b, y, p\}$ (for red, blue, yellow, and purple), if a node $a$ is colored with red, denoted by variable $a_r$, then the nodes with an edge from node $a$, denoted by $edge_{a,n_i}$ for node $n_i$, must be colored with distinct colors different from red. This can be modeled by

$$\neg a_r \vee \neg edge_{a,n_1} \vee \neg edge_{a,n_2} \vee ... \vee \neg edge_{a,n_m} \vee v_g$$

where $v_g$ denotes $allDiff(x_{n_1} : \{b, y, p\}, ..., x_{n_m} : \{b, y, p\})$.

In the language of $\mathrm{SAT}(gc)$, value variables may appear in clauses. This makes it more convenient to represent concepts related to CSP variables. E.g., the pigeonhole problem can be represented by an $allDifferent$ constraint where pigeons are CSP variables and holes are their domain values. One can express further that any solution should be such that pigeon-1 is either in the first hole or in the last hole. This can be represented by a disjunction of two value variables.

## IV. $\mathrm{SAT}(gc)$ Solver

We formulate a $\mathrm{SAT}(gc)$ solver in Algorithm 1, which is an extension of the iterative DPLL algorithm given in [10].

Given an instance $\Pi$ in $\mathrm{SAT}(gc)$, the solver first performs preprocessing by calling the function *gc_preprocess*() (Line 1, Alg. 1). It applies the standard preprocessing operations; however, it will not make any assignments on gc-variables. If *gc_preprocess*() does not solve the problem, then following a predefined decision heuristic the solver branches on an unassigned variable (Line 5, Alg. 1) to satisfy at least one clause in $\Pi$. Each decision variable is associated with a *decision level*, which starts from 1 and gets incremented on the subsequent decision level by 1. Then, the procedure gc_deduce() is invoked (Line 7, Alg. 1), and any new assignment generated by the procedure gets the same decision level of the current decision variable.

### A. Procedure *gc_deduce*()

In standard Boolean Constraint Propagation (BCP), there is only one inference rule, the *unit clause rule* (UCR). With the possibility of value literals to be assigned, either as part of a solution to a global constraint or as a result of decision or deduction, we need two additional propagation rules.

```
1  $status = gc\_preprocess()$
2  if $status = KNOWN$ then
3      return $status$
4  while $true$ do
5      $gc\_decide\_next\_branch()$
6      while $true$ do
7          $status = gc\_deduce()$
8          if $status == INCONSISTENT$ then
9              $blevel = current\_decision\_level$
10             $gc\_backtrack(blevel)$
11         else if $status == CONFLICT$ then
12             $blevel = gc\_analyze\_conflict()$
13             if $blevel == 0$ then
14                 return $UNSAT$
15             else
16                 $gc\_backtrack(blevel)$
17         else if $status == SATISFIABLE$ then
18             return $SATISFIABLE$
19         else
20             $break$
```

**Algorithm 1:** An Iterative Algorithm for SAT($gc$)

- **Domain Propagation (DP):** When a CSP variable $x$ is committed to a value $a$, all the occurrences of $x$ in other global constraints must also commit to the same value. Thus, for any global constraint $g$ and any CSP variable $x$ in it, whenever $x$ is committed to $a$, $Dom(x)$ is reduced to $\{a\}$. Similarly, when a value variable is assigned to false, the corresponding value is removed from the domain of the CSP variable occurring in any global constraint.
- **Global Constraint Rule (GCR):** If the domain of a CSP variable of a global constraint $v_g$ is empty, $v_g$ is not solvable, which is therefore assigned to false. If a global constraint $v_g$ is assigned to true, the constraint solver is called. If a solution is returned, the value variables corresponding to the generated solution are assigned to true; if no solution is returned, $v_g$ is assigned to false.

Now BCP consists of three rules, UCR, DP, and GCR, which are performed repeatedly until no further assignment is possible.

Since a global constraint $v_g$ in $\Pi$ is semantically equivalent to the disjunction of its solutions (in the form of value variables), when $v_g$ is assigned to false in the current partial assignment, the negation of the disjunction should be implied. Algorithm 1 does not do this explicitly. Instead, it checks the consistency in order to prevent an incorrect assignment.[4]

In case of $gc\_deduce()$ returning $INCONSISTENT$, the search backtracks to the current decision level (Line 10, Alg. 1). Otherwise, SAT($gc$) checks if a conflict has occurred. If yes, SAT($gc$) invokes $gc\_analyze\_conflict()$ (Line 12, Alg. 1), which performs conflict analysis, possibly learns a clause, and returns a backtrack level/point.

### *B. Procedure $gc\_analyze\_conflict()$*

We first recall DPLL based conflict analysis. The descriptions are based on the procedural process of performing (standard) BCP that implements what is called FirstUIP [11].

- *Antecedent clause (of a literal)*: the clause that has forced an implication on $l$.
- *Conflicting clause*: the first failed clause, i.e., the first clause during BCP in which every literal evaluates to false under the current partial assignment.
- *Conflicting variable*: The variable which was assigned last in the conflicting clause.
- *Asserting clause*: the clause that has all of its literals evaluate to false under the current partial assignment and has exactly one literal with the current decision level.
- *Resolution*: The goal is to discover an asserting clause. From the antecedent clause $ante$ of the conflicting variable and the conflicting clause $cl$ (see Alg. 2), resolution between the two combines $cl$ and $ante$ while dropping the resolved literals. This has to be done repeatedly until $cl$ becomes an asserting clause.
- *Asserting level*: the second highest decision level in an asserting clause. Note that by definition, an asserting clause has at least two literals.[5]

Similar to conflict analysis in [10], $gc\_analyze\_conflict()$ first finds a conflicting clause $cl$. Then it attempts to find an asserting clause using resolution, which is described by a while loop (Lines 2-31, Alg. 2). Inside the loop, it first obtains the last failed literal $lit$ in $cl$ by $choose\_literal(cl)$ (Line 3, Alg. 2). After that, it checks the literal $lit$.

(a) If $lit$ is a gc-literal, the conflict is due to the failure of the most recent call to the constraint solver for $lit$. There are two subcases.

  (1) If no previous DP operation was performed on the CSP variables in the scope of $lit$ and no call to $lit$ has succeeded before, then the failure of $lit$ is intrinsic, hence only the other literals in $cl$ may satisfy the clause. Thus, we drop $lit$ from $cl$ (Line 6, Alg. 2). There are three subcases.

    (i) If $cl$ is empty after dropping $lit$, the SAT($gc$) instance is not satisfiable (Line 9, Alg. 2).

    (ii) If $cl$ becomes unit after dropping $lit$, then $cl$ cannot be an asserting clause (by definition an asserting clause has at least two literals in it). So, we perform chronological backtracking, i.e., the current decision level is returned as the backtracking level (Line 12, Alg. 2).

[4] In our current treatment, this checking is performed at the end when a SAT($gc$) instance is solved. Then, the assignment of relevant value variables of a global constraint instantiates the global constraint which is then checked for consistency by a call to the CP solver.

[5] The process of resolution can produce a unit clause, in which case chronological backtracking is performed.

```
1  $cl = find\_conflicting\_clause()$
2  while $!isAsserting(cl)$ do
3      $lit = choose\_literal(cl)$
4      if $lit$ is a gc-literal then
5          if no DP is performed on the variables in the scope of $lit$ and $lit$ never has succeeded then
6              drop $lit$ from $cl$
7              if $cl$ is empty then
8                  $back\_dl = 0$
9                  return $back\_dl$
10             else if $cl$ is unit then
11                 $back\_dl = current\_decision\_level$
12                 return $back\_dl$
13         else
14             $dl = decision\_level(lit)$
15             if $lit$ is a decision literal then
16                 $back\_dl = dl - 1$
17                 return $back\_dl$
18             else
19                 $back\_dl = dl$
20                 return $back\_dl$
21     else
22         $ante = antecedent(lit)$
23         if $ante == NULL$ then
24             $back\_dl = backtrack\_point(lit)$
25             return back_dl
26         $cl = resolve(cl, ante, lit)$
27         $lit = choose\_literal(cl)$
28         $ante = antecedent(lit)$
29         if $ante == NULL$ and $lit$ is not a decision variable then
30             $back\_dl = backtrack\_point(lit)$
31             return back_dl
32 $add\_clause\_to\_database(cl)$
33 $back\_dl = clause\_asserting\_level(cl)$
34 return $back\_dl$
```

**Algorithm 2:** Conflict Analysis in SAT(gc)

(iii) Otherwise, continue with resolution.

(2) If the condition in (1) does not hold, i.e., $lit$ is not intrinsically unsolvable, then we perform chronological backtracking. If $lit$ is the *decision variable* of the current decision level, the previous decision level is returned as the backtrack level (Line 17, Alg. 2); Otherwise $lit$ is *forced* in the current decision level, in which case the current decision level is returned as the backtracking level (Line 20, Alg. 2).

(b) If $lit$ is not a gc-literal, it is then either a normal literal or a value literal. Any conflicting (non-decision) normal literal must have an antecedent clause, and a conflicting value literal may or may not have an antecedent clause, depending on how its truth value is generated.

If $lit$ has no antecedent clause (Line 23, Alg. 2), then it is a value literal assigned by a DP, which is triggered by a solution of a global constraint at the current decision level. In this case, SAT(gc) backtracks to the point where the corresponding global constraint is invoked for trying to generate an alternative solution for the same global constraint. The backtrack point is identified by the procedure $backtrack\_point(lit)$ (Line 24, Alg. 2).

In $gc\_analyze\_conflict()$, after the cases (a) and (b), inside the while loop, resolution is performed over $cl$ and $ante$ which results in a new $cl$. Notice that, the resulting clause $cl$ also has all of its literals evaluated to false, and is thus a conflicting clause. We then again check the last assigned literal $lit$ in $cl$. If $lit$ does not have any antecedent clause and $lit$ is not a decision variable, then it becomes the case of (b). Otherwise, this resolution process is repeated until $cl$ becomes an asserting clause, or either one of the above two cases (a) or (b) occurs. If an asserting clause is found, then the procedure $gc\_analyze\_conflict()$ learns the asserting clause $cl$ (Line 32, Alg. 2) and returns the asserting level as the backtracking level (Line 33, Alg. 2).

After $gc\_analyze\_conflict()$ returns a backtrack level, if it is 0 then SAT($gc$) returns UNSAT (Line 14, Alg. 1). Otherwise, it calls $gc\_backtrack(blevel)$ (Line 16, Alg. 1).

### *C. Procedure $gc\_backtrack(blevel)$*

The procedure $gc\_backtrack(blevel)$ distinguishes different types of conflict cases:

(a) If the backtracking level is obtained from an asserting clause, then the procedure $gc\_backtrack(blevel)$ backtracks to decision level $blevel$ and unassigns all the assignments up to the decision variable of $blevel + 1$. After backtracking the learned clause $cl$ becomes a unit clause and the execution proceeds from that point in a new search space within level $blevel$

(b) Otherwise, chronological backtracking is performed as follows:

(1) If the backtrack point is obtained from $backtrack\_point(lit)$, then $gc\_backtrack(blevel)$ backtracks and unassigns assignments up to that backtrack point in the current decision level.

(2) If conflict occurs because of a gc-literal fails to generate an alternative solution, then we backtrack to $blevel$ and unassign assignments up to the decision variable of $blevel$. [6]

(3) If inconsistency is detected during deduction, then $gc\_backtrack(blevel)$ performs backtracking similarly as in (b-(2)).

In case of (b-1), as the backtrack point is a gc-literal assignment, after backtracking SAT($gc$) attempts to generate another solution for the same gc-literal and the

[6] After dropping a failed gc-literal (Line 6, Alg. 2), if $cl$ becomes unit, then we return the current decision level (Line 11, Alg. 2) as the backtrack level. If that failed gc-literal is a *decision* literal, then we backtrack to the previous decision level of the current decision.

execution proceeds from that point. If no alternative solutions exist, it becomes a case of failed gc-literal. In other subcases of (b), after backtracking up to the decision variable of $blevel$, that decision variable is flipped. By flipping a variable, we mean to switch from one phase to the other for a normal variable or value variable, and switch from the current solution to the next one for a gc-variable; if a normal or value variable is already flipped or no further solutions for a gc-variable exist, then backtracking is meant to backtrack up to the decision variable of the preceding decision level (i.e., chronological backtracking).

### *D. Example*

Suppose, as a part of a $\mathrm{SAT}(gc)$ instance $\Pi$, we have

$$(c1)\ \neg r \vee d \quad (c2)\ r \vee v_g \quad (c3)\ t \vee s \vee \neg x_{1_a} \vee p \quad (c4)\ t \vee s \vee \neg x_{1_a} \vee \neg p$$

where $v_g$ is $allDiff(x_1 : \{a\}, x_2 : \{a, b\})$.

Let the current decision level be $dl$, and suppose at a previous decision level $dl'$ $\neg s$ and $\neg t$ were assigned, and at level $dl$ $\neg r$ is decided to be true. Then, $v_g$ is unit propagated from clause $c2$. The call for $v_g$ returns the CSP solution $\{x_1 = a, x_2 = b\}$, hence the value variables $x_{1_a}$ and $x_{1_b}$ are assigned to true; but then a conflict occurs on $c4$. So, $gc\_analyze\_conflict()$ is called.

In the procedure $gc\_analyze\_conflict()$, $c4$ (conflicting clause) and $c3$ (antecedent clause of the conflicting variable $p$) are resolved so that $cl$ becomes $t \vee s \vee \neg x_{1_a}$. Then, it is found that the last assigned literal in $cl$ is $\neg x_{1_a}$, which is generated by the solution of $v_g$. So, it returns the assignment point of $v_g$ as the backtrack point. The procedure $gc\_backtrack(blevel)$ unassigns all assignments up to $v_g$, and the constraint solver is called again, but this time $v_g$ generates no alternative solution. So, $v_g$ is assigned to $false$. Thus a conflict occurs on clause $c2$. Then $gc\_analyze\_conflict()$ is again called.

It is found that the conflicting variable is a forced gc-variable $v_g$ and a solution was previously generated for it. So, $gc\_analyze\_conflict()$ returns the current decision level as the backtracking level, and $gc\_backtrack(blevel)$ backtracks to the assignment $\neg r$, and flips it to $r$. This flipping immediately satisfies clause $c2$ and the literal $d$ is unit propagated from $c1$. The search continues from there. □

### *E. Soundness and Completeness of* $\mathrm{SAT}(gc)$ *Solver*

We claim Algorithm 1 is correct in the following sense:

> *Given a* $\mathrm{SAT}(gc)$ *instance* $\Pi$, *Algorithm 1 returns* $SATISFIABLE$ *if and only if* $\Pi$ *is satisfiable, and it returns* $UNSATISFIABLE$ *if and only if* $\Pi$ *is unsatisfiable.*

This assumes that the constraint solver is sound and complete, and terminating. A proof sketch of the statement can be constructed based on the following arguments.

- Algorithm 1 is terminating, as it traverses a finite search tree where the nodes are normal variables, value variables, and gc-variables. A gc-variable has only one phase when it is false (corresponding global constraint is not solvable) or it has one or more phases corresponding to one or more solutions when it is false. Then, the termination of the constraint solver guarantees a nite traversal of the search tree.
- The central argument regarding the correctness of Algorithm 1 is, if conflict analysis in Algorithm 2 does not involve any gc-variable, i.e., every "last failed literal" has an antecedent clause, then Algorithm 1 essentially reduces to the standard SAT solver employing the FirstUIP scheme in conflict analysis, which is known to be correct [11]. Otherwise, any learned clause involving a gc-literal is non-unit after dropping $lit$ from $cl$. In this case the failure can only be rescued by the rest of the literals in clause $cl$, i.e., this dropping does not change the satisfiability of $\Pi$. In all the other cases involving a gc-literal, backtracking is chronological so no satisfying assignment may be missed.
- Finally, if $SATISFIABLE$ is returned with an assignment $\theta$, then $\Pi$ is satisfiable. The only situation where this may not be the case is when value variable assignments in $\theta$ imply a solution of a gc-variable while $v_g$ is assigned to false in $\theta$. This is prevented by consistency checking.

## V. A Prototype Implementation of $\mathrm{SAT}(gc)$

We implemented a prototype called SATCP, by adopting ZCHAFF[7] as the DPLL engine and GECODE[8] as the CP solver.

The preprocessing function of ZCHAFF is modified slightly to avoid solving gc-variables in preprocessing. Intuitively, a solution of a gc-variable tends to make a good deal of implications by DP and EVAs. Thus, we implemented an eager variable selection heuristic, which gives higher priority on gc-variables of a SAT(gc) formula $\Pi$, according to the order in which they appear in $\Pi$. We modified the deduce function of ZCHAFF slightly to implement GCR, DP and EVAs. From the application perspective, a gc-variable occurs only in a unit clause positively. So GCR is implemented only for positive phased gc-literals. For conflict analysis and backtracking, we rely on the existing mechanisms of ZCHAFF, with slight modification. In SATCP, when a conflict occurs due to a gc-literal assignment, we raise a flag by assigning respective value to a designated flag variable. The conflict analyzer uses this flag to identify types of conflict that have occurred. As gc-literals are only assigned as decision literals, they cannot be forced. So, in case of any gc-literal conflict in SATCP, the conflict analyzer function returns $blevel$ and the backtracking function unassigns all the assignments up to the decision variable of $blevel$.[9]

CSP problems are modeled in GECODE by creating a subclass of a built-in class named Space and specifying the model inside that subclass. The solutions for a CSP model are searched by creating search engines [12]. From the CP perspective, every global constraint in a given $\mathrm{SAT}(gc)$ formula

[7] http://www.princeton.edu/ chaff/zchaff.html

[8] http://www.gecode.org/

[9] For gc-failure-conflict, $blevel$ is the previous decision level of the current decision and for value-variable-conflict, $blevel$ is the current decision level.

is an independent CSP problem. So, before starting executing SATCP, for every gc-variable $v_g$ in that formula, we create a subclass of the class Space, which models the global constraint as an independent CSP model. When SATCP is invokved, for each of the CSP models a search engine is created globally. SATCP uses the Branch and Bound (BAB) search engine, which is built-in in GECODE [12]. BAB search engine has a public method, named $next()$, which returns the next solution for the CSP model to which it is attached. When there is no more solution for the attached model, it returns $null$ and expires.

In SATCP, whenever a gc-variable $v_g$ is assigned, SATCP executes the $next()$ method of the search engine attached to the model of $v_g$ to get the next solution of $v_g$. When the attached search engine does not find any alternative solution for $v_g$, it returns $null$ and expires. If $v_g$ is assigned again (via backtracking), SATCP creates a new search engine at the local scope which searches for alternative solutions for $v_g$.

The reader is referred to [12], [13] for more details.

## VI. Experiments

We have carried out experiments with problems from puzzle domains and problems from planning domain. Here we report three of them as they are representative of some key aspects of our integration: the Latin square problem, the magic square problem, and the planning problem of block stacking with numerical constraints. The experiments were conducted in a UNIX server assembled with 2.10 GHZ Intel Core 2 Duo CPU (T5600) and 4GB RAM. For all the experiments, the cutoff time is 15 minutes, and the reported times are in second.

### *A. The Latin Square Problem*

A Latin square (LS) of order $n$ is an $n \times n$ array of $n$ numbers in which every row and columns must contain distinct numbers. We encode the constraint "no two numbers are assigned to the same cell" by negative binary clauses and the constraint "every number must appear exactly once in a row and in a column" by using $n$ *allDiff* global constraints, which are

$$\bigwedge_{k=1}^{n} allDiff(x_1^k, x_2^k, \ldots, x_i^k, \ldots, x_n^k)$$

where the domain of the CSP variables is $\{1 \ldots n\}$. The assignment $x_i^k = j$ $(1 \le i, j, k \le n)$ asserts that the number $k$ is placed on the $i^{th}$ row and $j^{th}$ column of the square.

We ran the LS problem on ZCHAFF (with SAT encoding) and on SATCP (with SAT($gc$) encoding) for instances of different sizes. The running times for ZCHAFF and for SATCP are given in Table I. Clearly, SATCP outperformed ZCHAFF.

To encode LS problem in SAT, the total number of clauses required are $O(n^4)$ [14], while a SAT($gc$) instance has $O(n^3)$ clauses. Thus, by the use of global constraints, the SAT($gc$) encoding is more compact. This plus the efficient propagators for $allDiff$ global constraint as implemented in GECODE is the main reasons for the better performance of SATCP.

| n | ZCHAFF | SAT($gc$) |
|---|---|---|
| 3 | 0 | 0 |
| 4 | 0 | 0 |
| 5 | 0 | 0 |
| 6 | 0.01 | 0.01 |
| 7 | 0.02 | 0.01 |
| 8 | 1.42 | 0.30 |
| 9 | 48.20 | 10.02 |
| 10 | — | 169.51 |

TABLE I
Experiments with Latin Square Problem.

| $n$ | SAT($gc$) |
|---|---|
| 3 | 0.00058 |
| 4 | 0.02 |
| 5 | 0.61 |
| 6 | 0.04 |
| 7 | 3.92 |
| 8 | — |

TABLE II
Experiments with NMS under monolithic constraint.

### *B. The Normal Magic Square Problem*

A normal magic square (NMS) of order $n$ is an arrangement of $n^2$ numbers, usually distinct integers, in a square, such that the sum of $n$ numbers in rows, columns, and in both diagonals are equal to a constant number. The constant sum over rows, columns and diagonals is called the *magic sum*, which is known to be $n(n^2+1)/2$.

We encode the NMS problem in two different encodings, referred to, respectively, as $monolithic$ encoding and $decomposed$ encoding. In the former the whole NMS problem is encoded by a single gc-variable, which is defined by a $allDiff$ constraint over $n^2$ cells (modeling the distinct constraint) and $2n{+}2$ $sum$ constraints each of which is over $n$ different cells. This encoding illustrates how CP is embedded in SAT. Notice that when SATCP runs monolithic encoding, all the variables and constraints are put inside the same constraint store.

Table II shows the experimental results for monolithic encoding. By this encoding we are able to solve the magic square problem up to size $7 \times 7$ by SATCP within 15 minutes (clearly, this is entirely due to the hours of GECODE). These results are consistent with the results of [15], as with this encoding SATCP runs much faster than the ASP solver CLASP, which uses aggregates to encode numerical constraints. For example, CLASP solves the NMS problem of order 7 in 450.58 seconds on a similar server. It also demonstrates that, the propagators of global constraints from CSP are more efficient than the aggregates in logic programming.

The decomposed encoding is contrasted with the monolithic encoding, where the monolithic constraint is decomposed into a collection of global constraints and SAT clauses. In this encoding, the $allDiff$ global constraint (of the monolithic encoding) is encoded by negative binary clauses and the sum constraints are encoded by $2n+2$ $sum$ constraints. With this decomposed encoding, SATCP solved NMS of order 3 in 3.17 sec. But for the instances of higher order, SATCP failed to generate a solution within 15 minutes.

In contratst to monolithic encoding, when SATCP runs the decomposed encoding, the related $sum$ global constraints are put into separate constraint stores. As a result, when a CSP variable $x$ is assigned to a value, the propagators for other related global constraints (those having $x$ on their scope) can not be executed. Therefore, the result of DP operation are not propagated to other related CSP variables. It results in an exponential number of enumerations of CSP solutions via a series of backtracking between the related global constraints. For this reason, SATCP performed poorly. In essence, poor propagation due to the processing of calls to different but related global constraints in different constraint stores is the reason of this inefficiency.

### *C. Block Stacking with Numerical Constraints*

In a table, there are $n$ stacks, each having $m_i$ blocks, where $1 \leq i \leq n$. Let $block_{ij}$ be the $j^{th}$ $(1 \leq j \leq m_i)$ block of $i^{th}$ $(1 \leq i \leq n)$ stack. In the initial configuration in every stack $i$ the first block $block_{i1}$ is placed on the table. If $m_i > 1$, then $block_{ij}$ is placed on $block_{i(j-1)}$. Every block $block_{ij}$ has a weight $w_{ij}$. We need to generate a plan for building a new stack of blocks by taking exactly one block from each of the initial $n$ stacks in such a way that two constraints hold, where *Constraint 1* is: "The total weight of the selected blocks should be equal to a certain total weight $W_0$", i.e., if block $j^1, j^2 \ldots j^n$ are selected respectively from stacks $1, 2 \ldots n$, then $w_{1j^1} + w_{2j^2} + \cdots + w_{ij^i} + \cdots + w_{nj^n} = W_0$; and *Constraint 2* is: "The block selected from the $i^{th}$ stack must be placed over the block selected from the $(i-1)^{th}$ stack."

Constraint 1 is encoded by using a $sum$ global constraint: $sum(stack_1, stack_2, \ldots, stack_n, W_0)$, where $Dom(stack_i)$ $= \{w_{i1}, w_{i2}, \ldots w_{im_i}\}$. The assignment $stack_i = w_{ij}$ asserts that the $j^{th}$ block form the $i^{th}$ stack is selected for building the goal stack.

From a STRIPS specification, we generate a planning instance that models constraint 2. We also introduce $m$ (where $m = m_1 + ... + m_n$) propositional variables, which are the value variables corresponding to the domain values of $stack_i$. Then for every pair of blocks, $block_{ij}$ and $block_{i'j'}$ (where $i' = i + 1$), we add their corresponding value literals to their stacking action (goal action) clauses at the goal layer. In our experiments, we use action based CNF encoding, generated by SATPLAN [16]. We have solved a number of instances with different numbers of blocks and stacks for this planning problem. The results are shown in Table III.

| Stacks:<br>Blocks<br>/Stack | Initial<br>Config./<br>Goal<br>Config. | Weights | $W_0$ | Time<br>(sec) |
|---|---|---|---|---|
| 2:2 | S1:(ON B A) S2:(ON D C)<br>/ (ON C B) | $w_A$=4,$w_B$=5<br>$w_C$=5,$w_D$=7 | 10 | 0.03 |
| 2:3 | S1:(ON B A) (ON C B)<br>S2:(ON E D) (ON F E)<br>/ (ON E B) | $w_A$=5,$w_B$=6<br>$w_C$=7,$w_D$=3<br>$w_E$=5,$w_F$=8 | 11 | 0.18 |
| 3:2 | S1:(ON B A) S2:(ON D C)<br>S3: (ON F E)<br>/ (ON E C) (ON C A) | $w_A$=3,$w_B$=4<br>$w_C$=2,$w_D$=4<br>$w_E$=5,$w_F$=2 | 7 | 0.29 |
| 3: 3 | S1:(ON B A) (ON C B)<br>S2:(ON E D) (ON F E)<br>S3:(ON H G) (ON I H)<br>/ (ON G D) (ON D A) | $w_A$=5,$w_B$=6<br>$w_C$=7, $w_D$=3<br>$w_E$=5, $w_F$=8<br>$w_G$=5, $w_H$=8<br>$w_I$=5 | 13 | 1.29 |
| 2:6 | S1:(ON B A) (ON C B)<br>(ON D C) (ON E D)<br>(ON F E)<br>S2:(ON G H) (ON I H)<br>(ON J I) (ON K J)<br>(ON L K)<br>/ (ON G A) | $w_A$=9,$w_B$=6<br>$w_C$=5,$w_D$=3<br>$w_E$=5,$w_F$=8<br>$w_G$=3,$w_H$=8<br>$w_I$=7,$w_J$=3<br>$w_K$=7,$w_L$=3 | 12 | 130.74 |

TABLE III
EXPERIMENTS WITH BLOCK STACKING WITH NUMERICAL CONSTRAINTS.

## VII. RELATED WORK

### *A.* SAT$(gc)$ *versus SMT*

Both adopt a DPLL based SAT solver as the overall solver. The embedded component of an SMT solver is a theory solver and for SAT$(gc)$ it is a constraint solver. The SMT solver uses a theory solver to determine the satisfiability of a portion of a $T$-formula. On the other hand, the SAT$(gc)$ solver uses a constraint solver to compute a solution of a global constraint for which the constraint solver is invoked. In SMT, whenever an inconsistent assignment is found by the $T$-solver, it informs the DPLL solver about the inconsistency and the $T$-solver sends information back to the DPLL solver as a theory lemma, so that the DPLL solver can learn a clause and backtrack to a previous point. On the other hand, in SAT$(gc)$ no such conflicting information is sent back from the constraint solver. The DPLL component of SAT$(gc)$ identifies the conflicts/inconsistencies related to the global constraint at hand and does the necessary domain setup for the respective CSP variables, clause learning and backtracking. The constraint solver is used as a black box, to solve the global constraints for which it is called.

### *B. Relation with Clingcon*

In [8], following the lazy SMT approach, a framework for integrating CSP style constraint solving in Answer Set Programming (ASP) has been developed, which is called CDNL-ASPMCSP. The ASP solver passes the portion of its (partial) Boolean assignment associated with constraints to a CP solver. The constraint solver checks the satisfiability of these constraint atoms. The call results either in a unsatisfiability signal or in an extension of the current partial assignment. For conflicting driven analysis, every inferred atom needs a reason from which it is inferred. As CP solver does not provide any reason for the solutions it generates, this approach constructs a non-trivial reason from the structural properties of the underlying CSP problem in the ASP program at hand. From this constructed reason, it finds a learned clause and backtracking level by using resolution based conflict analysis process. Currently, global constraints are not tightly integrated into CDNL-ASPMCSP. The semantic issue of allowing global constraints in non-monotonic rules is nontrivial. In contrast, SAT$(gc)$ is monotonic.

### *C.* SAT$(gc)$ *verses Lazy Clause Generation*

In [17] a tight integration of SAT and CP is presented, where domains of CSP variables are encoded by a given encoding scheme, namely *regular encoding*. Whenever a CSP variable is assigned a value by unit propagation of the attached SAT solver, CSP propagators corresponding to that assignment are invoked. Instead of returning reduced domains, the attached CSP solver returns some propagation rules, which emulate the functionality of those propagators. These rules are then converted into SAT clauses. The converted clauses are added one by one as learned clauses into the SAT clause database.

This approach requires heavy modification of the attached solvers. In our approach we can incorporate an off-the-shelf CSP solver more easily; when SAT is absent SAT$(gc)$ behaves like CSP. But for the solver described in [17], there is no guarantee.

### *D.* SAT$(gc)$*verses Universal Booleanization*

In [18] CSP instances are translated into SAT, where the translation is focused on one particular CSP modeling language called MiniZinc. This approach translates a MiniZinc CSP model into CNF, which is then solved by a SAT solver. It bundles MiniZinc and a SAT solver into one tool, named FznTini. First, the input MiniZinc model is Booleanized (not into CNF, but into a boolean version of the MiniZinc CSP model) into a language, which the authors have called *Booleanized FlatZinc*, by following a given translation approach for variables and constraints. The conversion of Booleanized FlatZinc version into DIMACS CNF is trivial. That DIMACS CNF instance is then solved by the SAT solver available in FznTini.

FznTini is a very specialized tool for solving MiniZinc CSP models by SAT solvers. Moreover, their translation scheme does not support global constraints.

## VIII. Conclusion and Future Work

We have developed an algorithmic framework, namely SAT$(gc)$ for embedding global constraints in a DPLL based SAT solver, in a tight fashion. We have shown that SAT$(gc)$ is more versatile than SAT or CP alone. From the CP point of view, SAT$(gc)$ supports conditional and disjunctive constraints in the same framework, and from the SAT point of view, certain types of structured information can be encoded by global constraints. We have seen that for the language where global constraints as well as value variables (Boolean variables representing CSP variables taking values) are allowed, the standard BCP is not enough, and we need to add three more deduction rules in addition to the unit clause rule. The language of SAT$(gc)$ also makes conflict analysis more involved. However, as we have seen, an integration of CP and SAT can make use of the existing mechanism of conflict-directed learning and backtracking in SAT, with additional cares to handle conflicts resulted from the presence of global constraints and value variables.

We implemented a prototype, called satCP , by adopting the SAT solver zchaff and CP solver gecode . Experiments are carried out for benchmarks from puzzle domains and planning domains, which leads to insights in compact representation, solving effectiveness, and novel usability of the new framework. A weakness of the current implementation of satCP is that calls to different global constraints are processed in different constraint stores, which results in poor propagation. To avoid this, it appears that the constraint solver needs to be modified so that the related propagators are called upon a CSP variable assignment by the DPLL component of satCP. Also, the search engines created at the beginning of the execution of satCP for each of the global constraints are required to be utilized for the failed gc-variables to avoid unnecessary overhead. Note that these weaknesses, though present in the current version of satCP, is not an intrinsic problem in SAT$(gc)$.

We plan to address these efficiency related issues in the further versions of satCP.

## References

[1] L. Bordeaux, Y. Hamadi, and L. Zhang, "Propositional satisfiability and constraint programming: A comparative survey." *ACM Computing Surveys*, vol. 38, no. 4, 2006.

[2] F. Rossi, P. van Beek, and T. Walsh, *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., 2006.

[3] W.-J. van Hoeve and I. Katriel, "Global constraints," in *Handbook of Constraint Programming*, F. Rossi, P. van Beek, and T. Walsh, Eds. Elsevier, 2006, ch. 7.

[4] D. Cohen, P. Jeavons, and P. Jonsson, "Building tractable disjunctive constraints," *Journal of the ACM*, vol. 47, no. 5, pp. 826–853, 2000.

[5] S. Mittal and B. Falkenhainer, "Dynamic constraint satisfaction problems," in *Proc. AAAI'90*, 1990, pp. 25–32.

[6] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "Solving SAT and SAT modulo theories: from an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T)," *Journal of the ACM*, vol. 53, no. 6, pp. 937–977, 2006.

[7] D. Chai and A. Kuehlmann, "A fast pseudo-boolean constraint solver," *IEEE Transactions on Computer-Aided Design of Intergrated Circuits and Systems*, vol. 24, no. 3, pp. 305–317, 2005.

[8] M. Gebser, M. Ostrowski, and T. Schaub, "Constraint answer set solving," in *Proc. ICLP'09*, 2009, pp. 235–249.

[9] M. Balduccini, "Industrial-size scheduling with asp+cp," in *Proc. LPNMR'11*, 2011, pp. 284–296.

[10] L. Zhang and S. Malik, " The quest for efficient Boolean satisfiability solvers," in *Proc. CAV'02*, ser. LNCS 2404. Springer, 2002, pp. 17–36.

[11] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik, "Efficient conflict driven learning in a boolean satisfiability solver," in *Proc. The 2001 IEEE/ACM international conference on Computer-aided design*, 2001, pp. 279–285.

[12] C. Schulte, G. Tack, and M. K. Lagerkvist, *Modeling and Programming with Gecode*, 2008.

[13] M. S. Chowdhury, "SAT with global constraints," Master's thesis, University of Alberta, 2011.

[14] I. Lynce, "Propositional satisfiability: Techniques, algorithms and applications," Ph.D. dissertation, Instituto Superior Tcnico, Universidade Tcnica de Lisboa, 2005.

[15] Y. Wang, J. You, F. Lin, L. Yuan, and M. Zhang, "Weight constraint programs with evaluable functions," *Annals of Mathematics and Artificial Intelligence*, vol. 60, no. 3-4, pp. 341–380, 2010.

[16] H. Kautz, B. Selman, and J. Hoffmann, "SatPlan: Planning as satisfiability," in *Abstracts of the 5th International Planning Competition*, 2006.

[17] O. Ohrimenko, P. J. Stuckey, and M. Codish, "Propagation via lazy clause generation," *Constraints*, vol. 14, no. 3, pp. 357–391, 2009.

[18] J. Huang, "Universal booleanization of constraint models," in *CP*, 2008, pp. 144–158.