

A System for Embedding Global Constraints into SAT

Md Solimul Chowdhury and Jia-Huai You

Department of Computing Science
University of Alberta

Abstract. Propositional satisfiability (SAT) and Constraint Programming (CP) are two dominant tools for solving combinatorial search problems. Both of these has their own strengths and weaknesses. In this report, we describe a tight integration of SAT with CP, called SAT(gc), which embeds global constraints into SAT. A system named SATCP is implemented by integrating the DPLL based 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. We highlight some issues with the current implementation of SATCP, with possible directions to resolve those issues.

Keywords: SAT, CSP, Global Constraints, Integration, Embedding.

1 Introduction

Constraint Programming (CP) [15] is a programming paradigm, developed for studying and solving constraint problems. It has been applied to solving many practical problems from domains of scheduling, planning, and verification [20]. For practical applications, languages for CP have been developed to facilitate the definitions of constraints in terms of primitive constraints and built-in constraints. One kind of these built-in constraints are called *global constraints* [19]. The use of global constraints not only facilitate problem representation, but also have very efficient implementation based on special data structures and dedicated constraint propagation mechanisms (see, for example, [3]).

Another way of solving combinatorial search problems is Boolean Satisfiability (SAT), in which a problem is represented by a collection of Boolean clauses, called a *formula*. To solve a SAT formula, we need to determine whether there is a truth value assignment that satisfies all the clauses.

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 [4]. A number of approaches have been pursued in this direction. For example, in SAT Modulo Theory (SMT) [13], 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., [5]). The paper [14] presents a tight integration of SAT and CP, where CSP propagators are emulated as learned SAT clauses. In [9], a framework for integrating CSP style constraint solving in Answer Set Programming (ASP) has been developed, employing an evaluation strategy similar to the lazy SMT approach. In [8], the authors propose a translational approach to constraint answer set programs and show its effectiveness. The usefulness of combining ASP and CP to industrial sized problems is demonstrated in [1].

We pursue a tight integration of SAT and CSP which implies tight interleaves between SAT and CSP solver. Tight integration poses a number of challenging issues like, how to represent a SAT problem in the presence of global constraints, and how deductions, conflict analysis and backtracking with learning can be performed in the presence of global constraints. In our work, we develop a framework to incorporate CSP style constraint solving into SAT solving with the anticipation that this tight integration will enhance the usability of SAT solver and increase its efficiency for some application domains. The report presented here is an extended version of [7], providing more details on implementation, some system level problems and possible solutions.

The rest of this report is organized as follows. The next section presents an embedding of global constraints into SAT, called SAT(gc). We describe a prototype system of SAT(gc), named SATCP, in Section 3. In Section 4, we describe the experiments we carry out, along with details on the benchmarks used, their encoding in SAT(gc), the experimental results and analysis. We then discuss some implementation issues of SATCP in Section 5. Section 6 compares SAT(gc) with related frameworks, with Section 7 providing conclusions and pointing to future directions.

2 The SAT(gc) Framework

In this section, we describe the framework for tightly embedding global constraints in SAT solving, which we refer to as SAT(gc).

Here, first we will provide the language and notation specification of the SAT(gc) framework. Then we will describe the algorithm for SAT(gc) solver, which deals with the following two major issues of the integration - how to perform deduction in the presence of global constraints in a SAT(gc) formula, and how to perform conflict directed backtracking and learning in the presence of global constraints.

2.1 Language and Notation

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. Propositions are also called *variables*. To distinguish, let us call these variables

normal variables. In the language of 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 SAT(gc) instance.

A gc-literal is true if and only if the corresponding global constraint is solvable. Then it 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 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 SAT(gc) instance, so that *unit propagation* enforces these axioms automatically.

With the language of SAT(gc) defined above, given a SAT(gc) instance, a gc-variable in it is semantically equivalent to a disjunction of conjunctions of value variables, augmented by the exclusive value axioms, with each conjunction representing a solution of the corresponding global constraint (if such a disjunction is empty, it represents *false*). That is, a SAT(gc) instance is semantically equivalent to a propositional formula. Given a SAT(gc) instance Π , 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 Π in the language of SAT(gc), determine whether there exists a variable assignment such that $\sigma(\Pi)$ evaluates to true.*

2.2 Representation in 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

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

In SAT(gc) this can be written by a clause of four gc-variables as

$$v_{g_1} \vee v_{g_2} \vee v_{g_3} \vee v_{g_4}$$

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 should be exactly one of sum constraints that holds, we can write

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

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

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

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

Apparently, SAT(gc) allows some complex interactions of constraints, e.g., conditional disjunctive constraints. Note that unlike typical CSP implementations (see, e.g., [2]), SAT(gc) encoding is capable of expressing conditional constraints and disjunctive constraints.

2.3 SAT(gc) Solver

We formulate a SAT(gc) solver in Algorithm 1, which is an extension of the iterative DPLL algorithm given in [23].

Given an instance Π in SAT(gc), the solver first performs preprocessing by calling the function *gc_preprocess()* (Line 1, Algorithm 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 pre-defined decision heuristic the solver proceeds to branch on an unassigned variable (Line 5, Algorithm 1) to satisfy at least one clause in Π . 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, Algorithm 1), and any new assignment generated by the procedure gets the same decision level of the current decision variable.

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.

- **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.

```

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 UNSATISFIABLE
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)

- **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. Note that, since all of these rules are monotonic, the order of their applications is unimportant.

Since a global constraint v_g in Π 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.¹ In case of *gc_deduce*() returning *INCONSISTENT*, the search backtracks to the current decision level (Line 10, Algorithm 1). Otherwise, SAT(gc) checks if a conflict has occurred. If yes, SAT(gc) invokes its conflict analyzer *gc_analyze_conflict*() (Line 12, Algorithm

¹ The general process of this checking can be highly complex, as it may involve the generation of all solutions of a global constraint. However, in many practical situations, e.g., for all the benchmarks we have experienced, this checking is not necessary.

1), which performs conflict analysis, possibly learns a clause, and returns a backtrack level/point.²

Conflict Analysis in SAT(gc) Let us first review some terms of DPLL based conflict analysis. The descriptions are based on the procedural process of performing (standard) BCP that implements what is called FirstUIP [22].

- *Antecedent clause (of a literal)*: the antecedent clause of a literal l is the clause which 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 , 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.

Below we describe the conflict analyzer *gc_analyze_conflict()* of SAT(gc) . For details on *gc_analyze_conflict()* see [7].

Like the conflict analyzer of [23], *gc_analyze_conflict()* attempts to find the asserting clause by using an iterative resolution process. In each iteration of the resolution process,

- it begins with a conflicting clause cl and obtains the last failed literal lit in cl . lit can be either a gc-literal, a value literal or a normal literal. The following cases handles each of the possibilities:
 - (a) lit is a gc-literal: Last call to the constraint solver for that lit failed. There are two subcases.
 - (1) lit is intrinsically unsolvable: No previous DP operation was performed on the CSP variables in the scope of lit and no call to lit has succeeded before; only the other literals in cl may satisfy the clause. We drop lit from cl . There are three subcases.
 - (i) cl becomes empty: The given SAT(gc) instance is not satisfiable.
 - (ii) cl becomes unit: Then cl cannot be an asserting clause (by definition an asserting clause has at least two literals in it). So, we perform chronological backtracking.

² A backtrack level leads to backtracking to the decision variable of that level, i.e., undoing all the assignments up to the decision variable of that level, while a backtrack point is a point of an assignment, which may or may not be a point of decision.

- (iii) Otherwise: Continue with resolution.
- (2) *lit* is not intrinsically unsolvable : We perform chronological backtracking. If *lit* is the *decision variable* of the current decision level, the previous decision level is returned as the backtracking level; Otherwise *lit* is *forced* in the current decision level, in which case the current decision level is returned as the backtracking level.
- (b) *lit* is a value literal: Value literal may or may not have an antecedent clause, depending on how its truth value is generated.
 - (1) *lit* has no antecedent clause: *lit* is a value literal assigned by a DP, SAT(gc) backtracks to the point where the corresponding global constraint (which triggered the DP operation) is invoked for trying to generate an alternative solution for the same global constraint.
 - (2) *lit* has an antecedent clause : Continue with resolution.
- (c) *lit* is a normal literal: Continue with resolution.

In *gc_analyze_conflict()*, after the cases (a) and (b), 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* and returns the asserting level as the backtracking level.

After *gc_analyze_conflict()* returns the backtracking level, if it is 0 then SAT(gc) returns UNSATISFIABLE (Line 14, Algorithm 1). Otherwise, it calls *gc_backtrack(blevel)*.

Backtracking in SAT(gc) The procedure *gc_backtrack(blevel)* distinguishes different types of conflict cases, depending on how the backtracking level is obtained:

- (a) *blevel* obtained from an asserting clause: *gc_backtrack(blevel)* backtracks to *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 the level *blevel*.
- (b) Otherwise: we perform chronological backtracking. (b) has three sub cases.
 - (1) *blevel* is a backtrack point in the current decision level: We backtrack and unassigns assignments up to that backtrack point in the current decision level.
 - (2) Conflict due to a gc-literal failure: We backtrack to *blevel* and unassign assignments up to the decision variable of *blevel*.
 - (3) Inconsistency is detected during deduction: We perform backtracking similarly as in (b-(2)).

Example 1. Suppose, as a part of a SAT(gc) instance Π , 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 a gc-variable for the global constraint, $allDiff(x_1 : \{a\}, x_2 : \{a, b\})$.

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 $x_2 = b$ 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 . Then, the constraint solver is called again, but this time v_g generates no alternative solution. So, v_g is assigned to false. As a result, a conflict occurs on clause $c2$. The procedure $gc_analyze_conflict()$ 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.

3 Implementation

We have implemented a prototype system of SAT(gc), which is called SATCP, where a SAT solver named ZCHAFF³ is used as the DPLL engine and a constraint solver named GECODE⁴ is used as the constraint solving engine.

3.1 Preprocessing, Deduction, Conflict Analysis and Backtracking

To comply with the SAT(gc) framework, in SATCP gc-variables are not assigned during the preprocessing step. Existing preprocessing function of ZCHAFF is modified accordingly. Intuitively, a solution of a gc-variable tends to make a good amount of implications by DP operation. This intuition has lead us to implement a variable selection heuristic, which puts higher priority on gc-variables of a SAT(gc) formula Π . The order in which gc-variables are considered for decision is determined by the order of appearance of gc-variables in Π . We have slightly modified the deduce function of ZCHAFF to implement GCR and DP. From the application perspective, a gc-variable occurs only in a unit clause positively. So GCR is implemented only for positive phased gc-literals. Before invoking GECODE for solving a gc-literal, according to the partial assignment, DP

³ <http://www.princeton.edu/~chaff/zchaff.html>

⁴ <http://www.gecode.org/>

operations are performed on the domain of its CSP variables. For conflict analysis and backtracking, we relied on the existing conflict analysis and backtracking function of ZCHAFF, with slight modification. In SATCP, when a conflict occurs because of 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, gc-literals can not 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*.⁵ For more details, please see [6].

3.2 GC-Variable and Search Engine

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 solution for a CSP model are searched by creating search engines [16].

From the CP perspective, every global constraint in a given SAT(gc) formula is an independent CSP problem. So, before starting executing SATCP, for every gc-variable v_{g_i} in that formula, we create a subclass of the class Space, which models the global constraint v_{g_i} as an independent CSP model. At the very beginning of the execution of SATCP, for each of the CSP models it creates a search engine globally.

SATCP uses the Branch and Bound (BAB) search engine, which is a built-in search engine of GECODE [16]. BAB search engine has a public method, named *next()*, which returns the next alternative solution for the CSP model to which it is attached. When there is no more alternative solution exists for the attached model, it returns *null* and expires. BAB engine also allows the implementation of a virtual function namely - *constraint()*, which allows us to post additional constraints before searching for the next alternative solution. This property of BAB engines is particularly favorable for our implementation. Before calling GECODE to generate an alternative solution for a gc-variable v_g , by using that *constraint()* function, SATCP posts a constraint named *dom*, which sets the domain of the CSP variables of v_g according to the current partial assignment. For more details on BAB search engine please see [16].

4 Experiments

In this section we present experiments performed with SATCP⁶ using three benchmark problems. We present these benchmarks, their SAT(gc) encodings, experimental results.

⁵ For gc-failure-conflict, *blevel* is the previous decision level of current decision level and for value-variable-conflict, *blevel* is the current decision level.

⁶ Source code of SATCP and benchmark instance generators can be found at <https://github.com/solimul/SATCP>

4.1 The Latin Square Problem

The Latin Square (LS) can be described as follows: *A Latin square of order n is an n by n array of n numbers (symbols) in which every row and columns must contain distinct numbers (symbols).*

We encode the constraint “no two numbers are assigned to the same cell” by negative binary clauses and constraint “every number must appear exactly once in a row and in a column” is encoded by using n *allDiff* global constraints.

We have run the LS problem on ZCHAFF (with SAT encoding) and on our prototype implementation system, SATCP (with SAT(gc) encoding) for instances of different sizes up to 10 by 10 LS. For the 10 by 10 instance of LS, ZCHAFF does not return any solution in 15 minutes, but SATCP returns it within 169.71 seconds⁷. For details, see [7].

To encode LS problem in SAT, total number of clauses required are $O(n^4)$ [11]. The SAT(gc) instance has $O(n^3)$ clauses. Thus, by the use of global constraints, one can write more compact representations in SAT(gc) than in SAT. This plus the efficient propagators for *allDiff* global constraint as implemented in GECODE seems to be the main reasons for the better performance of our implementation of SATCP.

4.2 The Normal Magic Square Problem

A Normal Magic Square (NMS) can be defined as follows: *A magic square 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 present two different encoding for the NMS problem. We refer the first one as *monolithic* encoding, where the whole NMS problem is encoded as a single gc-variable. The monolithic gc-variable is composed of a *allDiff* global constraint over n^2 cells to model the distinct constraint of the square and $2n + 2$ *sum* global constraints, each over n different cells of the square. We refer the second one as *decomposed* encoding, where the distinct constraint is encoded by negative binary clauses and the sum constraints are encoded by $2n + 2$ *sum* global constraints. In this encoding, each of the *sum* global constraints is encoded as single gc-variable. For both of the cases, the *sum* constraints encodes the sum constraints across the rows, columns and diagonals of the square.

Notice that in monolithic encoding all the variables and constraints are put inside the same constraint store, while in decomposed encoding the *sum* constraints are put into separate constraint store.

⁷ Size of 10 by 10 SAT(gc) instance : 1000 variables, 1210 clauses (1200 CNF clauses and 10 clauses containing 10 *allDiff* gc-variables)

We have solved NMS problem with monolithic constraint up to the size 7 by 7 by SATCP in under 4 seconds⁸. For the decomposed encoding, SATCP solved of order 3⁹ in 3.17 sec. But for the instances of higher order, SATCP failed to generate a solution within 15 minutes. For details, see [7]. In section 5.2. we present an analysis of the result with decomposed encoding of NMS.

The experimental results with the monolithic encoding of NMS running on SATCP confirms the results of [21], as with this encoding SATCP runs much faster than CLASP, which uses aggregates to encode numerical constraints. For example, CLASP solves the NMS problem of order 7 in 450.58 seconds. It also demonstrates that, the propagators of global constraints from CSP are more efficient than the aggregates from the logic programming framework.

4.3 The Planning Problem of Block Stacking with Numerical Constraints

The planning problem of block stacking with numerical constraints can be described as follows:

In a table, there are n ($n > 1$) stacks of blocks, each having m_i number of 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 have to generate a plan of actions for building a new stack of blocks by taking exactly one block from each of the initial n stacks in such a way that

- The total weight of the selected blocks should be equal to a certain total weight W_{max} . That is, if block $j^1, j^2 \dots j^n$ are selected respectively from stacks $1, 2 \dots n$, then $w_{1j^1} + w_{2j^2} + \dots + w_{ij^i} + \dots + w_{nj^n} = W_{max}$ (Constraint1).
- Block selected from the i^{th} stack must be placed over block selected from the $(i - 1)^{th}$ stack (Constraint2).

Constraint1 is encoded by using a *sum* global constraint as follows:

$$sum(stack_1, stack_2, \dots, stack_n, W_{max})$$

Where $Dom(stack_i) = \{w_{i1}, w_{i2}, \dots, w_{ij}, \dots, w_{im_i}\}$. 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 constraint2¹⁰. We also introduce $n * m$ (where $m = \bigwedge_{i=1}^n m_i$) propositional

⁸ Size of 7 by 7 monolithic SAT(gc) instance of NMS instance: 2401 variables, 50 clauses (49 CNF clauses and a clause for the monolithic gc-variable)

⁹ Size of 3 by 3 decomposed SAT(gc) instance of NMS: 81 variables, 664 clauses (657 CNF clauses and 8 clauses for 8 *linear* gc-variables.

¹⁰ For our experiment, we have used action based CNF encoding, generated by an automated SAT based planner named, SATPLAN.

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 goal layer).

We have solved five instances with different number of blocks and stacks for the planning problem of block stacking with numerical constraint by SATCP. We solve those problems under 131 seconds. For details, see [7].

5 Implementation Issues with SATCP

Here we briefly describe the implementation issues of SATCP.

5.1 Alternative Solution Generation for once failed gc-literal

Whenever a gc-variable v_{g_i} is assigned, SATCP executes the *next()* method of the search engine attached to the model of v_{g_i} to get the next alternative solution for that v_{g_i} . When the attached search engine does not find any alternative solution for v_{g_i} , it returns *null* and expires. If that v_{g_i} is assigned again, SATCP creates a new search engine at the local scope which searches for alternative solutions for v_{g_i} . Here, one point is worth mentioning. Every time an alternative solution needs to be generated for such a v_{g_i} (once failed), a local search engine needs to be created. So, for getting the j^{th} alternative solution for such v_{g_i} , $j-1$ solutions need to be generated. This is one reason, for which SATCP provides no solution within a reasonable amount of time for the decomposed encoding of NMS for the orders greater than 3.

This undesirable effect is caused by the inability of GECODE to reinstantiate the globally created search engine associated with a gc-literal (once failed), in case that gc-literal gets re-assigned by the SAT component. Apparently, to tackle this problem, GECODE needs to be modified to make the initially created global search engine reusable.

5.2 Inhibition of Constraint Propagation

In the decomposed encoding of NMS, we have used separate *sum* global constraints for modeling the sum constraints across each of the rows, columns and diagonals. Thus, at the GECODE end, we model each of these global constraints as separate CSPs. When SATCP executes on a decomposed magic square instance, these global constraints are put into separate constraint stores and are treated as independent and unrelated constraints. But, notice that these *sum* global constraints are related, as they are sharing CSP variables with each other. Assigning a CSP variable x to a value by any of the global constraints effects all CSP variables of the global constraints those have x on their scope. Though we are performing DP operation, after an assignment on a CSP variable x is made by a global constraint, the propagator for other related global constraint (those have x on their scope) are not getting notified of that assignment, as the

sum global constraints are put on different constraint store. Therefore, the result of DP operation cannot be propagated to other related CSP variables. This is another reason for which SATCP fails to provide any solution for decomposed encoding of NMS for the orders greater than 3.

Once a CSP variable is assigned to a value, SATCP is inherently incapable of propagating constraint on related global constraints, as each global constraints are put into separate constraint stores. To prevent this inhibition of constraint propagation, we need to put all the related global constraints in the same constraint store, with a call to a gc-literal returning solution only for itself. Modifying GECODE seems to be a way to tackle this problem.

6 Related Work

To compare SAT(gc) with SMT, both adopt a DPLL based SAT solver as the overall 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.

In [9], following the lazy SMT approach, a framework called CDNL-ASPMCSPP has been developed for integrating CSP style constraint solving in Answer Set Programming (ASP). The ASP solver passes the portion of its partial assignment associated with constraints to a CP solver, which checks the satisfiability of these constraint atoms. In case of conflicts, conflict analysis is performed as follows: it constructs a non-trivial reason from the structural properties of the underlying CSP problem in the ASP program at hand and use this reason to find a learned clause and backtracking level.

SAT(gc) is monotonic, in contrast to CDNL-ASPMCSPP, where the semantic issue of allowing global constraints in non-monotonic rules is nontrivial. SAT(gc) is more eager than CDNL-ASPMCSPP, in that whenever a Boolean literal associated with a constraint is assigned, it calls the constraint solver immediately. If any solution is returned, the current partial assignment is also eagerly extended by adding relevant value literals. In contrast to CDNL-ASPMCSPP, whenever a conflict involves a gc-literal or value literal, SAT(gc) performs chronological backtracking. The more eager approach of SAT(gc) identifies immediate occurrence of conflict due to a gc-literal or a value literal (if any exists) and enables SAT(gc) to perform chronological backtracking, as the backtracking points are obvious.

In [14] a tight integration of SAT and CP is presented. In their work, upon invocation, 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 [14], there is no guarantee.

An encoding scheme is presented in [17] to encode a finite linear CSP into SAT. The idea is to first convert a linear comparison into a primitive linear comparison by using the bounds of the comparison and then encode the primitive linear comparison into a SAT formula. In contrast to SAT(gc), which integrates SAT and CSP in a tight fashion, [17] presents a very specialized encoding scheme to solve finite linear CSPs by using SAT solvers.

In [10], the authors present a translation scheme from FlatZinc [12] CSP model into CNF, which is then solved by a SAT solver. [10] bundles MiniZinc and a SAT solver into one tool, named FznTini. Unlike SAT(gc), FznTini is not an extension of SAT, but it is a very specialized tool for solving FlatZinc CSP models by SAT solvers. Moreover, the translation scheme of FznTini does not directly support any global constraints.

In [18] the authors present a compact order encoding of CSP instances to SAT, which uses numeric systems of base ≥ 2 to represent an integer number, and the compact sized encoding results in fewer number of propagations than the previously proposed encoding in [17]. However, [18] does not address the encoding of other types of constraints, including global constraints. Also, unlike SAT(gc), it is not an extension of SAT.

7 Conclusion and Future Work

The current implementation of SATCP has two major weaknesses. Firstly, SATCP needs to perform repeated invocations to the constraint solver to generate alternative solutions for once failed gc-literal. Secondly, the assignments implied by the SAT solvers are not propagated into related global constraints as separate but related global constraints are needed to be kept into separate constraint stores. For both of these cases, apparently, the constraint solver needs to be modified. Note that these weaknesses, though present in the current version of SATCP, are not intrinsic problems of SAT(gc). We fully expect to address these problems in future versions of SATCP. Another interesting question is potential applications. A possible direction is the problem domain where planning and scheduling are tightly dependent on each other. Our benchmark from the planning domain shows a first step towards this direction.

References

1. Marcello Balduccini. Industrial-size scheduling with ASP+CP. In *Proc. LP-NMR'11*, pages 284–296, 2011.

2. P. Baptiste and C.L. Pape. Disjunctive constraints for manufacturing scheduling: principles and extensions. *International Journal of Computer Integrated Manufacturing*, 9(4):306–310, 1996.
3. Christian Bessiere, George Katsirelos, Nina Narodytska, Claude-Guy Quimper, and Toby Walsh. Propagating conjunctions of alldifferent constraints. In *Proc. AAAI’10*, 2010.
4. L. Bordeaux, Y. Hamadi, and L. Zhang. Propositional satisfiability and constraint programming: A comparative survey. *ACM Computing Surveys*, 38(4), 2006.
5. D. Chai and A. Kuehlmann. A fast pseudo-boolean constraint solver. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(3):305–317, 2005.
6. Md Solimul Chowdhury. SAT with Global Constraints. Master’s thesis, University of Alberta, 2011.
7. Md Solimul Chowdhury and Jia-Huai You. SAT with Global Constraints. In *ICTAI*, pages 73–80, 2012.
8. Christian Drescher and Toby Walsh. A translational approach to constraint answer set solving. *TPLP*, 10(4-6):465–480, 2010.
9. M. Gebser, M. Ostrowski, and T. Schaub. Constraint answer set solving. In *Proc. of the 25th International Conference on Logic Programming*, pages 235–249, 2009.
10. Jinbo Huang. Universal booleanization of constraint models. In *CP*, pages 144–158, 2008.
11. I. Lynce. *Propositional Satisfiability: Techniques, Algorithms and Applications*. PhD thesis, Instituto Superior Tecnico, Universidade Tecnica de Lisboa, 2005.
12. Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard cp modelling language. In *CP*, pages 529–543, 2007.
13. 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*, 53(6):937–977, 2006.
14. Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009.
15. F. Rossi, P. van Beek, and T. Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., 2006.
16. C. Schulte, G. Tack, and M. K. Lagerkvist. *Modeling and Programming with Gecode*. 2008.
17. Naoyuki Tamura, Akiko Taga, Satoshi Kitagawa, and Mutsunori Banbara. Compiling finite linear CSP into SAT. *Constraints*, 14(2):254–272, 2009.
18. Tomoya Tanjo, Naoyuki Tamura, and Mutsunori Banbara. A compact and efficient SAT-encoding of finite domain CSP. In *SAT*, pages 375–376, 2011.
19. W.-J. van Hoesve and I. Katriel. Global constraints. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 7. Elsevier, 2006.
20. Mark Wallace. Practical applications of constraint programming. *Constraints*, 1(1/2):139–168, 1996.
21. Yisong Wang, J. You, F. Lin, L. Yuan, and M. Zhang. Weight constraint programs with evaluable functions. *Annals of Mathematics and Artificial Intelligence*, 60(3-4):341–380, 2010.
22. 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*, pages 279–285, 2001.
23. L. Zhang and S. Malik. The quest for efficient Boolean Satisfiability solvers. In *Proc. CAV’02*, LNCS 2404, pages 17–36. Springer, 2002.