System Architecture and Implementation of a Prototyping Tool for SAT-based Constraint Programming Systems *

Takehide Soh[1], Naoyuki Tamura[1], Mutsunori Banbara[1], Daniel Le Berre[2], and Stéphanie Roussel[2]

[1] Kobe University 1-1, Rokko-dai, Nada, Kobe, Hyogo 657-8501 Japan
{soh@lion.,tamura@,banbara@}kobe-u.ac.jp
[2] CRIL-CNRS UMR 8188, Université d'Artois, SP-18, F-62307, Lens, France
{leberre,sroussel}@cril.univ-artois.fr

Abstract. This paper describes the Scarab system, a Scala implementation of a prototyping tool for developing SAT-based Constraint Programming systems. It consists of a Constraint Programming Domain-Specific Language, a SAT encoding module, and an interface to the back-end SAT solvers. The current implementation of Scarab uses Sat4j as the default back-end solver, thus runs on any platform for which a Java Virtual Machine exists. Scarab provides a rich modeling language embedded in Scala and enables programmers to rapidly specify problems and to experiment with different modelings. In Scarab, one can use integer variables and arithmetic constraints, and all of them are encoded into SAT without the need of developing a dedicated encoder. SAT solvers are then used for finding solutions.

Keywords: Propositional Satisfiability, Scala, Constraint Programming, Domain-Specific Language

1 Introduction

Propositional Satisfiability (SAT) is fundamental in solving many application problems in Artificial Intelligence and Computer Science [1]. Remarkable improvements in the efficiency of SAT solvers have been made over the last decade. Such improvements encourage programmers to develop SAT-based systems for logic synthesis, planning, scheduling, hardware/software verification, Constraint Satisfaction Problem (CSP) and so on. However, for a given problem, one usually has to develop a dedicated program which encodes it into SAT, and cannot focus on problem modeling which plays an important role in the system development process. It is therefore important to investigate modeling languages and development tools suited for SAT-based systems.

* This paper is an extended version of a paper entitled *Scarab: A Rapid Prototyping Tool for SAT-based Constraint Programming Systems (Tool Paper)*, which is accepted to SAT 2013.

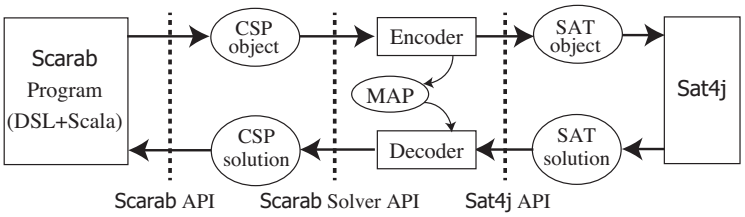

Fig. 1. Scarab Architecture

Scala[3] is a relatively new programming language receiving an increasing interest for developing real-world applications [2]. Scala is an integration of both functional and object-oriented programming paradigms. The main features of Scala are type inferences, higher order functions, immutable collections, and concurrent computation. It is also suitable for implementing Domain-Specific Language (DSL) [3] embedded in Scala. The Scala compiler generates Java Virtual Machine (JVM) bytecode, and Java class libraries can be used in Scala.

In this paper, we describe the Scarab[4] system, a Scala implementation of a prototyping tool for developing SAT-based Constraint Programming systems. It consists of a Constraint Programming DSL, a SAT encoding module, and an interface to the back-end SAT solvers. The current implementation of Scarab uses Sat4j [4] as the default back-end solver and runs on any platform for which a JVM exists. Fig. 1 shows an overview of Scarab architecture. First, a CSP object is defined through Scarab API from user's Scarab program. Second, when Scarab solver is called from the program, the CSP object is encoded to a SAT object. Sat4j is then called through Sat4j API from Scarab solver to find a SAT solution. Finally, a CSP solution is returned back to the user's program by decoding the SAT solution (if any).

The major design principles behind Scarab are providing an expressive, efficient, customizable, and portable workbench for SAT-based system developers.

Expressiveness: Scarab DSL provides a rich modeling language for Constraint Programming with the help of Scala. We show the expressiveness of Scarab through two examples: Square Packing and Latin Square. We also explain a way to realize the addition of extra constraints, constraint solving with assumptions, and commit/rollback of constraints with the help of Sat4j.

Efficiency: Scarab is efficient in the sense that it uses an optimized version of the order encoding [5] for encoding CSP into SAT. The order encoding has been used in an award-winning constraint solver Sugar [6].

Customizability: Scarab allows programmers to customize their own constraints and the search strategies. Scarab itself is compact. It is 500 lines long without comments and the core of encoding module is 25 lines long. This compactness makes it easier to implement other encoding modules.

[3] `http://www.scala-lang.org/`
[4] `http://kix.istc.kobe-u.ac.jp/~soh/scarab/`

```
 1: import jp.kobe_u.scarab.csp._
 2: import jp.kobe_u.scarab.solver._
 3: import jp.kobe_u.scarab.sapp._
 4:
 5: val n = 15; val s =36
 6:
 7: for (i <- 1 to n)
 8:   { int('x(i),0,s-i) ; int('y(i),0,s-i) }
 9: for (i <- 1 to n; j <- i+1 to n)
10:   add(('x(i) + i <= 'x(j)) || ('x(j) + j <= 'x(i)) ||
11:       ('y(i) + i <= 'y(j)) || ('y(j) + j <= 'y(i)))
12:
13: if (find) println(solution)
```

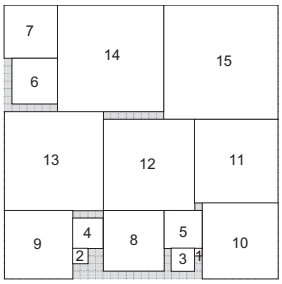

Fig. 2. Scarab Program of $SP(15, 36)$ and Solution

```
1: var n: Int = 5
2: for (i <- 1 to n; j <- 1 to n)  int('x(i,j),1,n)
3: for (i <- 1 to n) {
4:  add(alldiff((1 to n).map(j => 'x(i,j))))
5:  add(alldiff((1 to n).map(j => 'x(j,i))))
6:  add(alldiff((1 to n).map(j => 'x(j,(i+j-1)%n+1))))
7:  add(alldiff((1 to n).map(j => 'x(j,(i+(j-1)*(n-1))%n+1))))}
8:
9: if (find)  println(solution)
```

2	3	5	1	4
5	1	4	2	3
4	2	3	5	1
3	5	1	4	2
1	4	2	3	5

Fig. 3. Scarab Program of $LS(5)$ and Solution

Portability: The combination of Scarab and Sat4j makes it possible to develop portable applications which run on any platform with a JVM.

There have been several proposals providing Constraint Programming DSL for developing SAT-based systems: Copris (in Scala) [7], Numberjack (in Python) [8], Bee (in Prolog) [9], and B-Prolog (in Prolog) [10]. Among them, Copris and Numberjack are embedded in languages supporting functional programming features. Compared with them, Scarab is more compact, customizable, and tightly integrated with a SAT solver (i.e., Sat4j).

The rest of this paper is organized as follows. Section 2 presents two examples of Scarab programs. The classes and methods of Scarab are presented in Section 3. Section 4 presents some advanced solving techniques by using Sat4j. Section 5 shows experimental results of the Latin Square problem. The paper is concluded in Section 6.

2 Scarab Program Examples

We present two examples of Scarab programs. One is the square packing problem. Square Packing $SP(n, s)$ is a problem of packing a set of squares of sizes 1×1 to $n \times n$ into an enclosing square of size $s \times s$ without overlapping. For a given $SP(n, s)$, the most direct modeling would be using integer variables $x_i \in \{0, \ldots, s-i\}$ and $y_i \in \{0, \ldots, s-i\}$ for each square i $(1 \leq i \leq n)$ such that each pair $(x_i,\ y_i)$ represents the lower left coordinates of the square i. We then enforce the constraint $(x_i + i \leq x_j) \vee (x_j + j \leq x_i) \vee (y_i + i \leq y_j) \vee (y_j + j \leq y_i)$ to ensure that there is no overlapping for any distinct squares i and j $(1 \leq i < j \leq n)$.

Fig. 2 shows a Scarab program of $SP(15, 36)$. Scarab DSL can be used to concisely express the above modeling. The first three lines import classes provided by Scarab. The `int` method defines integer variables in line 7–8. The notation `'x` denotes the symbol x in Scala. Symbols `'x(i)` and symbols `'y(i)` with indices are converted to integer variable objects x_i and y_i with the help of implicit conversion of Scala. The `add` method defines non-overlapping constraints in line 9–11. The `find` method searches a solution after encoding the defined CSP to SAT, and the `solution` method returns the solution in line 13.

Another example is the Latin Square problem used in international CSP solver competition [11]. Latin Square $LS(n)$ is a problem of placing different n numbers into $n \times n$ matrix such that each number is occurring exactly once in each row, each column, and each modular diagonals in two directions. For a given $LS(n)$, we use a $n \times n$ matrix of integer variables $x_{i,j} \in \{1, \ldots, n\}$ $(1 \leq i, j \leq n)$. The exact one constraints can be expressed by using *alldiff* constraints [12] that is one of the best known and most studied global constraints in constraint programming [13].

Fig. 3 shows a Scarab program of $LS(5)$. Scarab DSL can be used to concisely express the *alldiff*-based modeling with the help of `map` method of Scala. For example, `alldiff((1 to n).map(j => 'x(i,j)))` in line 4 corresponds to the constraint $\mathit{alldiff}\,(x_{i,1}, x_{i,2}, \ldots, x_{i,n})$ in each row i.

3 Classes and Methods of Scarab

This section explains the classes/objects and methods provided by Scarab. They can be classified into:

- Classes/objects for constraint modeling defined in `jp.kobe_u.scarab.csp` package (such as `Term`, `Constraint`, and `CSP` classes),
- Classes/objects for constraint solving defined in `jp.kobe_u.scarab.solver` package (such as `Encoder`, `SatSolver`, and `Solver` classes), and
- Classes/objects implementing Scarab DSL defined in `jp.kobe_u.scarab` package (such as `Scarab` class).

3.1 Classes and Methods for Constraint Modeling

The following classes for linear arithmetic constraints are provided in Scarab and the diagram of these classes is shown in Fig. 4. Note that fields and methods are omitted from the figure except the classes of `CSP`, `Domain`, and `Assignment`.

- `Term` is an abstract class for linear arithmetic expressions. Its subclasses include `Var` (integer variables) and `Sum` (linear arithmetic expressions).
- `Constraint` is an abstract class for linear arithmetic constraints. Its subclasses include `Literal` (literals), `And` (conjunction of constraints), and `Or` (disjunction of constraints). `Literal` is an abstract class whose subclasses include `Bool` (positive Boolean literals), `Not` (negative Boolean literals), and `LeZero` (linear arithmetic comparisons).

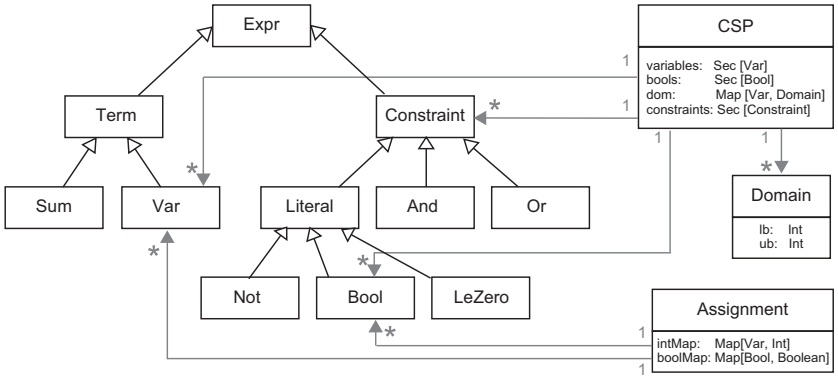

Fig. 4. Class Diagram of Classes for Constraint Modeling

– `CSP` is a class for CSPs. Other related classes include `Domain` class for representing domains of integer variables and `Assignment` class for representing assignments on integer variables and Boolean variables.

`Term` classes: `Term` is an abstract class for linear arithmetic expressions. `Term` objects are constructed with usual arithmetic operators as defined in the following BNF where *Int*, *String*, *Seq*, and *Any* are categories of integers, strings, sequences, and any objects of Scala language respectively.

$$
\begin{aligned}
T ::= {} & V \mid \texttt{-}\ T \mid T\ \texttt{+}\ \mathit{Int} \mid T\ \texttt{+}\ T \mid T\ \texttt{-}\ \mathit{Int} \mid T\ \texttt{-}\ T \mid T\ \texttt{*}\ \mathit{Int} \mid \\
& \texttt{Sum(}V, \ldots\texttt{)} \mid \texttt{Sum(Seq(}V, \ldots\texttt{))} \\
V ::= {} & \texttt{Var(}\mathit{String}, \mathit{String}, \ldots\texttt{)} \mid V\texttt{(}\mathit{Any}, \ldots\texttt{)}
\end{aligned}
$$

When the above syntax is used, any terms are converted to linear arithmetic expressions of the form $b + \sum a_i x_i$,and expressed by `Var` or `Sum` objects.

– `Var(name: String, indices: String*)`
Class for integer variables is implemented as a case class `Var`. The `name` field is its head name, and `indices` field is a (possibly empty) sequence of its indices. Two integer variables are equal when their names and indices are all equal. New integer variable can be constructed by adding extra indices to existing integer variable. It is implemented by defining `apply` method of `Var` class. Any objects including integers can be used as extra indices. The following example illustrates how to create integer variable objects.

```
scala> val x = Var("x")   // Creates a new integer variable x
scala> val x1 = x(1)      // Creates a new integer variable x(1)
scala> val y = Var("y")   // Creates a new integer variable y
scala> val y12 = y(1,2)   // Creates a new integer variable y(1,2)
```

– `Sum(b: Int, coef: Map[Var,Int])`
Class for linear arithmetic expressions is implemented as a case class `Sum`.

The `b` field is the constant part, and the `coef` field is the mapping from integer variables to their coefficient values. For example, the object `Sum(-1, Map(x -> 1, y -> 2))` represents a linear arithmetic expression $-1+x+2y$. The following example illustrates how to create `Sum` objects.

```
scala> x + y * 2 - 1                // −1 + x + 2y
scala> Sum((1 to 3).map(i => x(i))) // x(1) + x(2) + x(3)
```

`Constraint` classes: `Constraint` is an abstract class for linear arithmetic constraints. `Constraint` objects are constructed with usual arithmetic operators as defined in the following BNF.

C ::= B | T *op* T | `!` C | C `&&` C | C `||` C |
`And(`C, ...`)` | `And(Seq(`C, ...`))` | `Or(`C, ...`)` | `Or(Seq(`C, ...`))`
op ::= `<=` | `<` | `>=` | `>` | `===` | `!==`
B ::= `Bool(`*String*, *String*, ...`)` | B`(`*Any*, ...`)`

When the above syntax is used, any constraints are converted to negation normal form which is constructed from literals, conjunctions, and disjunctions. Note that, as is shown in the example of Latin Square, Scarab programs are not restricted to this syntax, we can combine it with Scala program.

- `Bool(name: String, indices: String*)`
  Class for Boolean variables is implemented as a case class `Bool`. The `name` field is its head name, and `indices` field is a (possibly empty) sequence of its indices. Two Boolean variables are equal when their names and indices are all equal. New Boolean variable can be constructed by adding extra indices to existing Boolean variable in the same way as integer variables.
- `Not(p: Bool)`
  Class for negative Boolean literals is implemented as a case class `Not`. The `p` field is its Boolean variable.
- `LeZero(sum: Sum)`
  Class for linear arithmetic comparisons is implemented as a case class `LeZero`. It represents the comparison of `sum` ≤ 0. Equality and disequality relations are translated to conjunction and disjunction of `LeZero` objects respectively. The following example illustrates how to create `LeZero` objects.

  ```
  scala> x > y      // 1 − x + y ≤ 0
  scala> x === y    // x − y ≤ 0 ∧ −x + y ≤ 0
  scala> x !== y    // 1 + x − y ≤ 0 ∨ 1 − x + y ≤ 0
  ```

- `And(cs: Constraint*)`
  Class for conjunctions is implemented as a case class `And`. The following example illustrates how to create `And` objects.

  ```
  scala> Bool("p") && x > 0             // p ∧ 1 − x ≤ 0
  scala> And((1 to 2).map(i => x(i)>0)) // 1 − x(1) ≤ 0 ∧ 1 − x(2) ≤ 0
  ```

- `Or(cs: Constraint*)`
  Class for conjunctions is implemented as a case class `Or`.

`CSP` **classes:** In Scarab, CSP is implemented as `CSP` class and provides methods for adding integer variables, Boolean variables, and constraints. `Domain` class is provided for representing domains of integer variables and `Assignment` class is provided for representing assignments on integer variables and Boolean variables.

- `CSP(var variables: Seq[Var], var bools: Seq[Bool], var dom: Map[Var,Domain], var constraints: Seq[Constraint])`
  Class for CSPs is implemented as a case class `CSP`. The `variables` field represents the current list of integer variables, and the `bools` field represents the current list of Boolean variables. The `dom` field is used to remember the current mapping from integer variables to their domains. The `constraints` field represents the current list of constraints. The following methods are provided to modify `CSP` objects.
  - `int(x: Var, d: Domain)` adds the integer variable `x` and its domain `d` to the CSP.
  - `int(x: Var, lb: Int, ub: Int)` adds the integer variable `x` and its domain as `Domain(lb, ub)` to the CSP.
  - `bool(p: Bool)` adds the Boolean variable `p` to the CSP.
  - `add(c: Constraint)` adds the constraint `c` to the CSP.
  - `show` displays the CSP to standard output.
  - `commit` remembers the current setting of the CSP. Only one commit point is allowed.
  - `rollback` resets the setting of the CSP to the last commit point.

  The following example illustrates how to create and modify `CSP` object.

```
scala> val csp = CSP()
scala> csp.int(x, 1, 9)  // x ∈ {1..9}
scala> csp.add(x !== 2)  // x ≠ 2
```

- `Domain(lb: Int, ub: Int)`
  Class for domains of integer variables is implemented as a case class `Domain`. Only interval domains are allowed in the current implementation. The `lb` and `ub` fields are its lower and upper bound values respectively.
- `Assignment(intMap: Map[Var,Int], boolMap: Map[Bool,Boolean])`
  Class for assignments of integer variables and Boolean variables is implemented as a case class `Assignment`. `Assignment` objects are used to represent the CSP solutions.

3.2 Classes and Methods for Constraint Solving

The following classes for SAT encoding and solving CSP are provided in Scarab and the diagram of these classes is shown in Fig. 5.

- `Encoder` is an abstract class for encoding CSP to SAT. `OrderEncoder` class is the only implementation in the current version. `Simplifier` class is used for Tseitin transformation [14] by `OrderEncoder`.
- `SatSolver` is an abstract class for SAT solvers. `Sat4j` class described in Section 4 is the only implementation in the current version.
- `Solver` is a class for solving CSP. It calls `Encoder` to encode the given CSP to SAT, and calls `SatSolver` to find a solution.

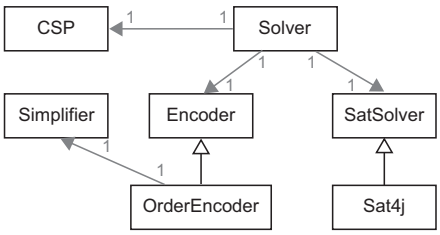

Fig. 5. Class Diagram of Classes for Constraint Solving

`Encoder` classes:

- `Encoder(csp: CSP, satSolver: SatSolver)`
  Class for encoding the given CSP to SAT is provided as an abstract class `Encoder`. The `csp` field specifies the CSP to be encoded, and the `satSolver` field specifies the SAT solver to which the encoded CNF is passed. `Encoder` is also used to decode back the solution obtained by the SAT solver to a CSP solution. The following methods and fields are provided by this class.
  - `encodeCSP` method encodes the given CSP to SAT. It calls `encode(x)` method for each integer variable `x` and `add(c)` method for each constraint `c`, and obtained SAT clauses are passed to the SAT solver through the `addAllClause` method.
  - `encode(x: Var)` is an abstract method obtaining the list of encoded SAT clauses for the integer variable `x`. The method is implemented in `OrderEncoder` class.
  - `add(c: Constraint)` is an abstract method to encode the constraint `c`, and implemented in `OrderEncoder` class. It translates the constraint to CNF formula (a list of literals) by Tseitin transformation, encodes the CNF formula to SAT clauses, and the SAT clauses are passed to the SAT solver.
  - `decode` method returns the CSP solution as an `Assignment` object by decoding the SAT solution obtained from the SAT solver. It calls `decode(x)` method for each integer variable `x`
  - `decode(x: Var)` method is an abstract method to obtain the value of the integer variable `x` from the SAT solution. It is implemented in `OrderEncoder` class.
- `OrderEncoder(csp: CSP, satSolver: SatSolver)`
  `OrderEncoder` is the only implementation of `Encoder` class in the current version of Scarab.

`SatSolver` classes:

- `SatSolver`
  Class for SAT solvers is provided as an abstract class `SatSolver`. The methods provided by this class include the followings.

```
 1:  val csp = CSP()
 2:  val x = Var("x")
 3:  csp.int(x(1), 1, 3)     // x(1) ∈ {1..3}
 4:  csp.int(x(2), 1, 3)     // x(2) ∈ {1..3}
 5:  csp.add(x(1) < x(2))    // x(1) < x(2)
 6:
 7:  val satSolver = new Sat4j()
 8:  val encoder =
 9:     new OrderEncoder(csp, satSolver)
10:  val solver =
11:     new Solver(csp, satSolver, encoder)
12:  while (solver.find)
13:    println(solver.solution)
```

```
1:  import
2:   jp.kobe_u.scarab.sapp._
3:
4:  int('x(1), 1, 3)
5:  int('x(2), 1, 3)
6:  add('x(1) < 'x(2))
7:
8:  while (find)
9:    println(solution)
```

Fig. 6. Example Programs for $x(1) < x(2)$, where $x(1), x(2) \in \{1..3\}$. The left side is written in Scala using Scarab classes. The right side is written by using Scarab DSL.

  - `addClause(lits: Seq[Int])` is an abstract method to add the SAT clause to the SAT solver.
  - `addAllClause(clauses: Seq[Seq[Int]])` is an abstract method to add the SAT clauses to the SAT solver.
  - `isSatisfiable` is an abstract method to search a SAT solution.
  - `model` is an abstract method returning the SAT solution when the given SAT instance is found satisfiable by `isSatisfiable` method.

– `Sat4j`
  `Sat4j` is the only implementation of `SatSolver` class in the current version. It is explained in Section 4.

`Solver` class:

– `Solver(csp: CSP, satSolver: SatSolver, encoder: Encoder)`
  Class for CSP solvers is provided as a class `Solver`. It consists of CSP, SAT solver, and encoder. The methods provided by this class include the followings.

  - `find` method searches a CSP solution by encoding the CSP with `encodeCSP` method of the encoder, searching a SAT solution with `isSatisfiable` method of the SAT solver, and decoding back to a CSP solution with `decode` method of the encoder. It searches the next CSP solution when it is called again by adding the negation of the last CSP solution as extra constraint.
  - `find(assumption: Constraint)` method searches a CSP solution under specified assumption. The constraint given as the assumption should be translated to a conjunction of SAT literals.
  - `solution` method returns the CSP solution as an `Assignment` object when the given CSP instance is found satisfiable by `find` method.

  The left side of Fig. 6 illustrates how to use the `Solver` class.

3.3 Classes and Methods Implementing Scarab DSL

`jp.kobe_u.scarab.sapp` object provides several functions useful as Constraint Programming DSL. The `sapp` object contains `CSP`, `Sat4j`, `OrderEncoder`, and `Solver` objects as its default CSP, SAT solver, encoder, and CSP solver respectively. It provides the following methods by delegating to the underlying objects.

- `int(x: Var, lb: Int, ub: Int)`, `bool(p: Bool)`, `add(c: Constraint)` `commit`, `rollback`
- `find`, `find(assumption: Constraint)`, `solution`

It provides implicit conversion from Scala symbols to integer variables (`Var` objects). The right side of Fig. 6 illustrates how to use `sapp` object. We can see that the program is more concisely written.

3.4 Implementation of the *alldiff* constraint

Global constraints such as *alldiff* play an important role in Constraint Programming [13]. In Scarab, all we have to do for implementing global constraints is just decomposing them into simple arithmetic constraints. The effectiveness of this approach has been recently shown by Bessiere *et al* [15]. For example, the following is a naive implementation of the *alldiff*$(x_1, \ldots, x_n)$ constraint:

```
def alldiff(xs: Seq[Var]) =
  And(for (Seq(x, y) <- xs.combinations(2)) yield x !== y)
```

In the above, *alldiff*$(x_1, \ldots, x_n)$ is decomposed into pairwise not-equal constraints $\bigwedge_{1 \le i < j \le n}(x_i \neq x_j)$. It is also known that some additional hints such as permutation constraints and pigeon hole constraints can be effective for performance improvement. The following is an optimized implementation of the *alldiff*$(x_1, \ldots, x_n)$ constraint:

```
def alldiff(xs: Seq[Var]) = {
  val lb = for (x <- xs) yield csp.dom(x).lb
  val ub = for (x <- xs) yield csp.dom(x).ub
  val ph =                                                  // pigeon hole
    And(Or(for (x <- xs) yield !(x < lb.min+xs.size-1)),
        Or(for (x <- xs) yield !(x > ub.max-xs.size+1)))
  def perm =                                                // permutation
    And(for (num <- lb.min to ub.max)
        yield Or(for (x <- xs) yield x === num))
  val extra = if (ub.max-lb.min+1 == xs.size) And(ph,perm) else ph

  And(And(for (Seq(x, y) <- xs.combinations(2)) yield x !== y),extra)
}
```

In the above, for each *alldiff* constraint, two extra pigeon-hole constraints $\neg\bigwedge(x_i < lb + n - 1)$ and $\neg\bigwedge(x_i > ub - n + 1)$ are added, where lb and ub are the lower and upper bounds of $\{x_1, x_2, \ldots, x_n\}$. Extra permutation constraints $\bigwedge_{i=lb}^{ub}\bigvee_{j=1}^{n}(x_j = i)$ are also added if $n = ub - lb + 1$. Other global constraints such as *element*, *weightedsum*, *cumulative*, and *global cardinality* can be implemented in the same way as done in Sugar [6].

4 Advanced Solving Techniques using Sat4j

Scarab adopts Sat4j [4] as its default implementation of the `SatSolver` class. By using the features of Sat4j, Scarab provides the functions: addition of extra constraints; CSP solving with assumptions; commit and rollback of constraints. These functions can be used to implement advanced solving techniques in Scarab, such as the search of optimum solution and enumeration of the solutions.

Incremental search: As illustrated in the example below, Scarab allows to add extra constraints after finding a solution.

```
scala> int(x, 1, 3); int(y, 1, 3)   // x ∈ {1..3},  y ∈ {1..3} is added
scala> add(x === y)                 // x = y is added
scala> find                         // solution is x = 3,  y = 3
scala> add(x !== 3)                 // x ≠ 3 is added as extra constraint
scala> find                         // solution is x = 2,  y = 2
```

In the first call of `find` method, the whole CSP is encoded and generated SAT clauses are added to Sat4j, then `isSatisfiable` method is called to find a solution. In the second call of `find` method, only the extra constraint $x \neq 3$ is encoded and added to Sat4j, then `isSatisfiable` method is called again. The learned clauses obtained by the first `find` are kept at the second call. Note that addition of extra constraints is not allowed after the CSP becomes unsatisfiable.

Search under assumptions: `find(assumption: Constraint)` method provides CSP solving under assumption given by the specified constraint. The constraint of assumption should be encoded to a conjunction of literals (otherwise an exception is raised). Then, the literals are passed to Sat4j through `isSatisfiable(literals)` method which enables SAT solving under assumption. The following example (continuation of the last example) illustrates the usage of solving with assumptions.

```
scala> find(y === 3)          // UNSAT
scala> find(x === 1)          // solution is x = 1,  y = 1
```

Commit and Rollback of Constraints: Commit and rollback of constraints are implemented as follows.

1. `commit` saves the current state (the number of integer variables, Boolean variables, and constraints) of CSP. Only one commit point is allowed in the current implementation.
2. `rollback` restores the state of CSP to the last commit point. It also initializes the state of Sat4j using its `reset` method. The whole CSP is encoded and added to Sat4j again at the next call of `find` method.

```
 1: val lb = n; var ub = s; int('m, lb, ub)
 2:
 3: for (i <- 1 to n)
 4:   add(('x(i)+i <= 'm) && ('y(i)+i <= 'm))
 5:
 6: while (lb <= ub && find('m <= ub)) {
 7:   add('m <= ub); ub -= 1
 8: }
 9:
10: while (find)
11:      println(solution)
```

Fig. 7. Decremental Search by Assumption

```
 1: var  lb = n; var ub = s; commit
 2:
 3: while (lb < ub) {
 4:   var size = (lb + ub) / 2
 5:   for (i <- 1 to n)
 6:     add(('x(i)+i<=size)&&('y(i)+i<=size))
 7:   if (find) {
 8:     ub = size; commit
 9:   } else {
10:     lb = size + 1; rollback
11:   }
12: }
```

Fig. 8. Binary Search by Commit/Rollback

Commit/rollback methods provides much more flexible control on solving. However, the encoding process is repeated again, and obtained learned clauses are lost in the subsequent solving after the CSP becomes unsatisfiable.

The following example (continuation of the last example) demonstrates the usage of commit/rollback.

```
scala> commit                // commit point is made
scala> add(x < y)            // x < y is added
scala> find                  // UNSAT
scala> rollback              // x < y is dropped
scala> find                  // solution is x = 2,  y = 2
```

Search of Optimum Solution: We demonstrate the application of the above three techniques for searching the optimum solution of a given CSP. An example of Square Packing in Fig. 2 is used to find the minimum size of enclosing square.

In Fig. 7, a decremental search is implemented by using assumptions and the addition of extra constraints. In the program, integer variable $m \in \{lb, \ldots, ub\}$ denotes the size of the enclosing square. We can run this program by adding it to the bottom of the program of Fig. 2. In line 6, `find` method is called under $m \leq ub$. If a solution exists, a constraint $m \leq ub$ is added and ub is decremented (line 7). In line 10 and 11, all optimal solutions are enumerated. In Scarab, when

Table 1. Benchmark Results on $LS(n)$: CPU time of Scarab

n	3 UNSAT	4 UNSAT	5 SAT	6 UNSAT	7 SAT	8 UNSAT	9 UNSAT
alldiff (naive)	0.164	0.153	0.183	0.398	0.210	T.O.	T.O.
alldiff (optimized)	0.230	0.209	0.236	0.264	0.221	0.212	0.235

n	10 UNSAT	11 SAT	12 UNSAT	13 SAT	14 UNSAT	15 UNSAT	16 UNSAT
alldiff (naive)	T.O.	0.347	T.O.	T.O.	T.O.	T.O.	T.O.
alldiff (optimized)	0.370	0.332	0.981	0.545	9.792	389.917	458.187

`find` is called for a CSP without any change, it searches the next CSP solution by adding the negation of the last CSP solution as extra constraints.

Fig. 8 shows a binary search by using commit/rollback methods and the addition of extra constrains. Line 4 computes the size which divides the region of the current lower and upper bounds in half. Lines 5 and 6 add extra constraints that limits the size of the enclosing square. In line 7, `find` is called. If the CSP is satisfiable, the upper bound is updated and the addition of extra constraints is committed (line 8). Otherwise, the lower bound is incremented and the addition of extra constraints is canceled. Note that the whole CSP is encoded again only after the CSP becomes unsatisfiable.

5 Experiments

To evaluate the basic efficiency of Scarab, we carried out experiments on the Latin Square problem shown in Fig. 3. We use two different implementations of *alldiff* presented in Section 3.4. We set a timeout (T.O) of 1 hours. All times were measured on Mac OS X with Xeon 2.93 GHz and 2 GB memory for JVM.

We note that $LS(n)$ with $n > 8$ were unsolvable by any solver in the 2009 CSP solver competition except Sugar which solved the problem with up to $n = 12$.

Table 1 shows CPU time of Scarab in seconds for solving $LS(n)$ of $3 \leq n \leq 16$. Scarab with an optimized implementation of *alldiff* succeeded in solving the problem with up to $n = 16$. The easy-to-customize feature of Scarab contributes the performance improvement for this problem.

6 Conclusion

This paper presents the Scarab system, a Scala implementation of a prototyping tool for developing SAT-based Constraint Programming systems. As is shown by two examples of Square Packing and Latin Square, Scarab DSL provides a rich modeling language for constraint programming with the help of Scala.

By using the features of Sat4j, Scarab provides the functions: incremental search; CSP solving with assumptions; commit and rollback of constraints. These functions can be used to implement advanced solving techniques in Scarab, such as searching for optimum solutions and enumerating those solutions.

An interesting extension is to introduce more features from Sat4j. For instance, Sat4j allows to compute a Minimal Unsatisfiable Subformula (MUS) in case of inconsistency. Sat4j also provides a built-in optimization framework and a specific handling of solution enumeration. Sat4j also handles natively cardinality and pseudo-Boolean constraints. Another extension would be to allow the modeling of soft constraints in the DSL, i.e. constraints which may be violated against a penalty. The source code and information of Scarab is available in `http://kix.istc.kobe-u.ac.jp/~soh/scarab/`.

References

1. Biere, A., Heule, M., van Maaren, H., Walsh, T., eds.: Handbook of Satisfiability. In Biere, A., Heule, M., van Maaren, H., Walsh, T., eds.: Handbook of Satisfiability. Volume 185 of Frontiers in Artificial Intelligence and Applications (FAIA)., IOS Press (2009)
2. Odersky, M., Spoon, L., Venners, B.: Programming in Scala. Second edn. Artima, Inc. (2010)
3. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Comput. Surv. **37**(4) (2005) 316–344
4. Le Berre, D., Parrain, A.: The sat4j library, release 2.2. JSAT **7**(2-3) (2010) 59–6
5. Tamura, N., Taga, A., Kitagawa, S., Banbara, M.: Compiling finite linear CSP into SAT. Constraints **14**(2) (2009) 254–272
6. Tamura, N., Tanjo, T., Banbara, M.: System description of a SAT-based CSP solver Sugar. In: Proceedings of the 3rd International CSP Solver Competition. (2008) 71–75
7. : Copris. `http://bach.istc.kobe-u.ac.jp/copris/`
8. Hebrard, E., O'Mahony, E., O'Sullivan, B.: Constraint programming and combinatorial optimisation in NumberJack. In: Proceedings of the 7th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2010), LNCS 6140. (2010) 181–185
9. Metodi, A., Codish, M.: Compiling finite domain constraints to SAT with BEE. Theory and Practice of Logic Programming **12**(4-5) (2012) 465–483
10. Zhou, N.F.: The language features and architecture of b-prolog. Theory and Practice of Logic Programming **12**(1-2) (2012) 189–218
11. Lecoutre, C., Roussel, O., van Dongen, M.R.C.: Promoting robust black-box solvers through competitions. Constraints **15**(3) (2010) 317–326
12. Régin, J.C.: A filtering algorithm for constraints of difference in CSPs. In: Proceedings of the 12th National Conference on Artificial Intelligence (AAAI 1994). (1994) 362–367
13. van Hoeve, W.J., Katrie, I.: Global constraint. In: Handbook of Constraint Programming. Foundations of Artificial Intelligence. Elsevier (2006) 169–208
14. Tseitin, G.S.: On the complexity of derivations in the propositional calculus. Studies in Mathematics and Mathematical Logic Part II (1968) 115–125
15. Bessiere, C., Katsirelos, G., Narodytska, N., Quimper, C.G., Walsh, T.: Decompositions of all different, global cardinality and related constraints. In: Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009). (2009) 419–424