Scala 上で実現されたSAT型制約プログラミングシステムのための開発ツール

宋 剛秀　番原 睦則　田村 直之

命題論理の充足可能性判定 (SAT) 問題を解くプログラムである SAT ソルバーが 2000 年以降大きく進歩しており，活発に開発と研究が行われている．このような背景から，様々な分野で SAT ソルバーを利用した SAT 型システムが成功をおさめるようになった．通常 SAT 型システムには，与えられた問題を SAT 問題へと符号化する専用のプログラムが必要になるが，この開発コストのために制約モデルや符号化方法など問題を解くうえで重要な役割を担う部分に注力できないという問題があった．本論文では SAT 型システムのための開発ツールである Scarab を提案する．Scarab は制約プログラミングのためのドメイン特化言語, SAT 符号化モジュール, SAT ソルバーへのインターフェースから構成されており，SAT 型制約プログラミングシステム開発者を対象に，表現性，変更容易性，効率性を備えたワークベンチを提供することを目的としたツールである．Scarab はオブジェクト指向言語と関数型言語が融合された Scala 言語上にコンパクトに実装されている．バックエンドの SAT ソルバーに Java で実装された Sat4j を用いることにより，インクリメンタル解法等の高度な解法を容易に実現できる点も特長の一つである．

1　はじめに

命題論理の充足可能性判定 (**SAT**; propositional satisfiability) 問題は計算理論において中心的であり，また人工知能や計算機科学の分野においても重要な問題である [4] [13]．2000 年以降 SAT 問題を解くためのプログラムである **SAT ソルバー**の性能が飛躍的に向上しており [22]，このような性能向上は与えられた問題を SAT 問題に符号化し，SAT ソルバーを用いて解を求める **SAT 型システム**の開発を促進してきた．これまで論理合成，プランニング，スケジューリング，ハードウェア・ソフトウェアの検証などで SAT 型システムが成功をおさめている [13] [3] [33] [21]．SAT 型システムは効果的な方法であるが，与えられた問題に対して提案した解法を毎回はじめから実装するのは効率が悪い．そこで制約充足問題 (**CSP**; constraint satisfaction problem) などの汎用的に問題を記述して解くことができる **SAT 型制約プログラミングシステム**が開発されてきた [31] [34] [12] [20] [40]．例えば Sugar [31] は専用の制約言語で制約充足問題 (**CSP**; constraint satisfaction problem) を記述し，順序符号化 [30] によって SAT 問題へと符号化した後に SAT ソルバーによって解を求める SAT 型制約プログラミングシステムである．2008 年と 2009 年に CSP ソルバー競技会 [18] のグローバル制約部門で優勝し，また近年では 2013 年にパッキング配列の最良解を更新するのに用いられた [23]．

このように Sugar をはじめとする SAT 型制約プログラミングシステムは高い求解性能を実現しつつある．しかし一方で高度な SAT 型解法を容易に実現できるような機能性についてはあまり注目がされていない．例えば，近年 Counterexample Guided Abstraction Refinement (CEGAR) [5] のモデル検査以外への応用や，SAT ソルバーに SAT 節以外の制約を組込む，組込み制約を用いた SAT 型解法が提案されている [15] [16] [8] [6] [28]．Sugar は専用の記述言語を用いることでユーザにとって理解し易い制約表現を可能にしているが，一方でループなどの汎用プログラミング

Prototyping Tool for SAT-based Constraint Programming Systems in Scala

Takehide Soh, Mutsunori Banbara, Naoyuki Tamura, 神戸大学情報基盤センター, Information Science and Technology Center, Kobe University.

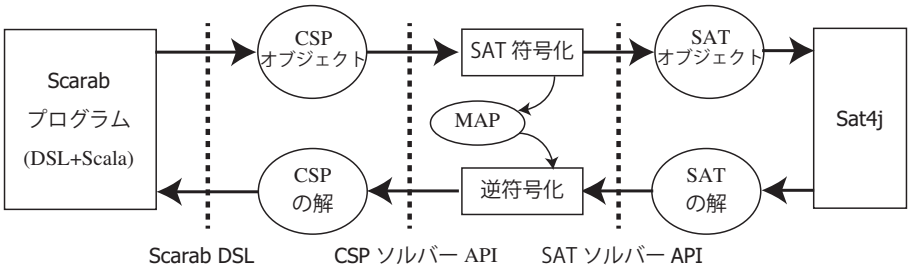

図 1　Scarab の構成

言語で提供されている制御構造を扱うことができず CEGAR を直接記述できない．また SAT ソルバーとの連携はファイルを介して外部プロセスを起動することによって行なうなど SAT ソルバーをブラックボックスとして扱っている．これはいろいろな種類の SAT ソルバーをコストをかけずに切り替えることが可能であり，利点にもなるが SAT ソルバー内部の低レベル機能の利用と拡張には対応することが難しくなる．

上記のような機能性の議論に加えて，使い易い SAT 型制約プログラミングシステムには与えられた問題を簡単に記述できる表現性も重要である．**Scala** は Martin Odersky により設計された比較的新しいプログラミング言語である [25]．特長として関数型言語とオブジェクト指向言語の融合と**ドメイン特化言語** (**DSL**; Domain-specific Language) を実装するために適した機能を数多く備えていることが挙げられる．また 2006 年の ACM Symposium on Programming Languages and Systems (POPL) の Martin Odersky の招待講演において Scala のアプリケーションとして制約プログラミングが示唆されており [24]，制約記述のためのドメイン特化言語を構築するのに適した言語の一つであると考えられる．

本論文では高度な SAT 解法を実現するためのワークベンチとして，Scala 上で実装された SAT 型制約プログラミングシステム Scarab[†1] を提案する．Scarab は制約プログラミングのためのドメイン特化言語である Scarab DSL, SAT 符号化モジュール，そしてバックエンドの SAT ソルバーへのインターフェースから構成される．SAT 符号化には Sugar に採用されている順序符号化 [30] の他に組込み制約を用いた符号化を利用可能である．また Scarab の大きな特長の一つは SAT ソルバー Sat4j [17] との密な連携である．共に Java 仮想マシン (JVM) 上で動作し，Scarab から API を通じて Sat4j の低レベル機能を自由に利用することができる．また外部プロセスによる起動を許容すれば Sat4j 以外の SAT ソルバーも利用可能である．これらの外部 SAT ソルバー用の API も備えており，連言標準形で表される命題論理式 (CNF 式) が記述されたファイルを入力とする SAT ソルバーを Scarab から利用することができる．

Scarab の設計方針は SAT 型システム開発者に表現性，効率性，変更性，可搬性を備え，SAT 符号化と SAT ソルバーをグラスボックス化したワークベンチを提供することである．

表現性:　Scarab DSL と Scala の両方を用いて与えられた問題を記述することが可能である．

効率性:　Scarab は最適化された順序符号化法を標準で用いているという点で効率的である [30]．順序符号化は 2008 年，2009 年に CSP ソルバー競技会のグローバル部門で優勝した Sugar [31] に採用されている．

変更性:　Scarab では SAT 型システム開発者が自前の制約を定義し，変更・改良することが可能である．また Scarab のソースコードは全体で 1000 行ほどであり，Scarab 本体の変更も可能である．特に順序符号化の中心部分の実装は 20 行ほどで行われている．

可搬性:　Scarab と Sat4j は両方とも JVM 上で

†1 http://kix.istc.kobe-u.ac.jp/~soh/scarab/

動作し，可搬性のあるシステムを実現可能である．

図 1 に Scarab の構成図を示す．まず SAT 型システム開発者が Scala と Scarab DSL を用いて記述したプログラム (Scarab プログラム) によって CSP オブジェクトが生成される．次に Scarab プログラムによって CSP ソルバーが求解のために API を通して呼ばれた時に，CSP オブジェクトは SAT オブジェクトへと変換される．続いて CSP ソルバーから SAT ソルバー Sat4j が API を通して実行される．解が存在すれば逆符号化を通して CSP の解が返される．

汎用プログラミング言語を制約記述に用いる SAT 型制約プログラミングシステムの従来研究として Copris (in Scala) [34], Numberjack (in Python) [12], Bee (in Prolog) [20], B-Prolog (in Prolog) [40] が挙げられる．この中でも, 著者らが開発した Copris は Scala 上の埋込み DSL を備えている点で Scarab と同様のツールである．Copris では Sugar を SAT 符号化モジュールとして用いており，SAT ソルバーを外部プロセスとして起動するなど SAT ソルバーとの連携は疎に行われる．一方，Scarab は，SAT 符号化，SAT ソルバーのグラスボックス化を設計方針としており，SAT 符号化モジュールを含めて Scala 上で 1000 行程度で実装されている．これによりコードはコンパクトで変更容易であり，新しいアイデアを実験し易い．また SAT ソルバーとの密な連携によりインクリメンタル解法や組込み制約など SAT ソルバーの低レベル機能の利用ができる点，それらを Scala 上で拡張できる点において Copris と異なるなるツールになっている．

以下では図 1 の流れに従って Scarab を説明する．2 節では制約表現の部分にあたる Scarab プログラムと二つの問題記述例を説明する．次に与えられた制約集合を SAT 問題の節に符号化する方法を 3 節で述べる．4 節は SAT 問題に符号化された制約集合をどのようにして SAT ソルバーで解くかについて説明を行い，Scarab の特長である Sat4j を用いた高度な解法について説明を行う，5 節では性能評価を示す．

2 Scarab プログラム

Scarab を利用するための最も基本的な方法は Scarab プログラムを記述することである．Scarab プログラムの記述には Scala と制約プログラミングのための DSL である Scarab DSL の両方を利用可能である．本節では Scala と Scarab DSL を説明した後，二つの問題記述例を通して Scarab の基本的機能の説明を行う．

2.1 Scala の概要

Scala は[†2] Martin Odersky により設計された比較的新しいプログラミング言語で，Twitter の分散 DB フレームワークに用いられたことなどもあり，近年注目を集めている [25]．言語上の特長としては，関数型言語とオブジェクト指向言語の融合，強力な型推論，型安全性，高階関数，不変コレクションなどが挙げられる．

処理系としては，JVM (Java Virtual Machine) へのコンパイラとインタラクティブな実行環境 (REPL; Read Eval Print Loop) が用意されている．Java との親和性は高く，Java のクラス・ライブラリをそのまま利用できる．

さらに Scala は，関数型言語およびオブジェクト指向言語としての高度な記述能力，リスト，マップ，集合等の豊富なコレクションフレームワーク，演算子の多重定義や柔軟な構文，オブジェクトのメソッドのインポート機能など，埋込みの **ドメイン特化言語** (**DSL**; Domain-specific Language) を実装するために適した機能を数多く備えている [19]．特に Scarab DSL ではケースクラス，暗黙変換，シングルトンオブジェクトおよびそのオブジェクトからのインポートなどの機能を利用しているが，これらの詳細については文献 [25] を参照されたい．

2.2 Scarab DSL

Scarab DSL は Scala 上の埋込み DSL (embedded DSL) として実装されている．CSP における項 ($x+y$ 等の数式) および制約 ($(x > 0) \vee (y > 0)$ 等の論理式) を Scala 中で表現するために，次のようなクラスを定義する: `Term` (項を表すクラス), `Var` (整数変数を表すクラス), `Bool` (ブール変数を表すクラス), `Constraint` (制約を表すクラス)．これらのクラスは

†2 `http://www.scala-lang.org/`

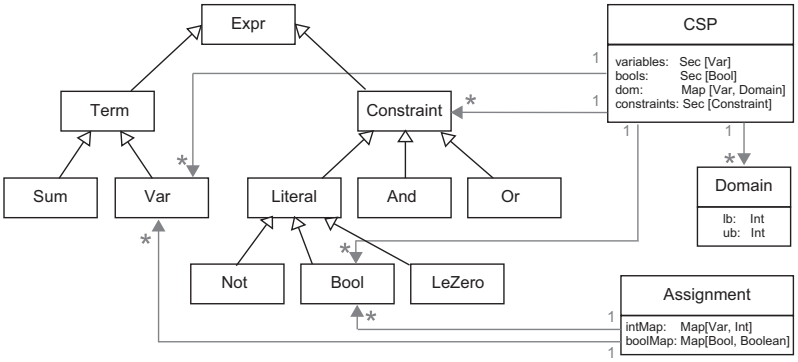

図 2　Scarab における CSP クラス図

$$
\begin{aligned}
T &::= V \mid \texttt{-}\ T \mid T\ \texttt{+}\ \mathit{Int} \mid T\ \texttt{+}\ T \mid T\ \texttt{-}\ \mathit{Int} \mid T\ \texttt{-}\ T \mid T\ \texttt{*}\ \mathit{Int} \mid \texttt{Sum}(V, \ldots) \mid \texttt{Sum}(\mathit{Seq}[V]) \\
V &::= \texttt{Var}(\mathit{String}, \mathit{String}, \ldots) \mid V(\mathit{Any}, \ldots) \\
C &::= B \mid T\ \mathit{op}\ T \mid \texttt{!}\ C \mid C\ \texttt{\&\&}\ C \mid C\ \texttt{||}\ C \mid \\
&\qquad \texttt{And}(C, \ldots) \mid \texttt{And}(\mathit{Seq}[C]) \mid \texttt{Or}(C, \ldots) \mid \texttt{Or}(\mathit{Seq}[C]) \mid \texttt{alldiff}(\mathit{Seq}(T, \ldots)) \\
\mathit{op} &::= \texttt{<=} \mid \texttt{<} \mid \texttt{>=} \mid \texttt{>} \mid \texttt{===} \mid \texttt{!==} \\
B &::= \texttt{Bool}(\mathit{String}, \mathit{String}, \ldots) \mid B(\mathit{Any}, \ldots)
\end{aligned}
$$

図 3　Scarab における制約の構文規則

jp.kobe_u.scarab.csp パッケージに属している．上記のクラスの関係を表したクラス図を図 2 に示す．

図 3 に Scarab DSL における制約の構文規則を示す．ここで T，V，C，B は上述の Term，Var，Constraint，Bool クラスを表す．また *Int*，*String*，*Seq* をそれぞれ Scala における整数 (Integer)，文字列 (String)，列 (Sequence) クラスとする．*Any* は全てのクラスの親クラスとする．論理演算子は|| と Or が論理和，&& と And が論理積を表している．加えて比較演算子として<= ($\leq$), <, >= ($\geq$), >, === ($=$), !== ($\neq$) を用意している．

Scarab DSL における制約定義の例を Scala の REPL の実行例を通して説明する (import jp.kobe_u.scarab.csp._ の実行を仮定している)．整数変数を表すクラス Var は，変数名に加え文字列のリストを添字として指定可能である．また，すでにある Var オブジェクトに任意の添字を追加した新しい Var オブジェクトを作成できるように，apply メソッドを定義している．

```
scala> val x = Var("x")
x: jp.kobe_u.scarab.csp.Var = x
scala> x(1,2)    // x.apply(1,2) と同じ
res0: jp.kobe_u.scarab.csp.Var = x(1,2)
```

これらの整数変数を用いた制約 $x \leq 1 \vee x_{1,2} \leq 1$ の記述例を以下に示す．

```
scala> x <= 1 || x(1,2) <= 1
res1: jp.kobe_u.scarab.csp.Or =
 Or(LeZero(Sum(-1+x)),LeZero(Sum(-1+x(1,2))))
```

ここで注意されたいのは Scarab では制約中の線形制約は定義されるとまず $\sum_i a_i x_i - c \leq 0$ (但し a_i は非零の整数 c は整数値，x_i は整数変数) の形に変換されるということである．線形制約のクラスは LeZero である．上記の例では $x \leq 1 \vee x_{1,2} \leq 1$ は $x - 1 \leq 0 \vee x_{1,2} - 1 \leq 0$ に変換される．

この他に Scarab では Scala の暗黙変換を利用してシンボル (' で始まる記号) を整数変数として利用できるようになっており，次のように簡潔に制約を記述できる．

```
scala> 'y <= 0 || 'y(1,2) <= 0
```

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

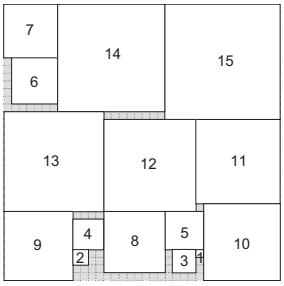

図 4 $SP(15, 36)$ **の Scarab プログラムと計算された解**

```
res2: jp.kobe_u.scarab.csp.Or =
 Or(LeZero(Sum(+y)),LeZero(Sum(+y(1,2))))
```

最後に Scarab DSL を用いた CSP の定義例を以下に示す.

```
// CSP の生成
scala> val csp = CSP()
// 整数変数 x, y ∈ {1, ..., 3} の追加
scala> csp.int('x, 1, 3)
scala> csp.int('y, 1, 3)
// 制約 x + y ≤ 2 の追加
scala> csp.add('x + 'y <= 2)
```

このように定義された CSP の解を計算するために CSP ソルバーのクラス `Solver` を用意している. `Solver` は `jp.kobe_u.scarab.solver` パッケージに属しており, 定義された CSP の解を計算する `find` メソッドを提供している. `find` は与えられた CSP を SAT に符号化し, SAT ソルバーを呼び出すことで解の計算を行う. `Solver` クラスおよび関係クラスについては後述する 4 節の図 8 も参考にされたい.

2.3 Scarab プログラム例

本節では Scala と Scarab DSL を用いたプログラム (Scarab プログラム) の例を説明する. 本節のプログラム例では次のインポート文の実行を仮定している.

```
import jp.kobe_u.scarab.csp._
import jp.kobe_u.scarab.solver._
import jp.kobe_u.scarab.sapp._
```

ここで `jp.kobe_u.scarab.sapp` は Scarab のアプリケーションのためのシングルトンオブジェクトであり, CSP, CSP ソルバーそれぞれのデフォルトオブジェクトや, それらに対する `int`, `add`, `find` 等のメソッドを提供している. これによりデフォルトオブジェクトに対する操作をオブジェクト名を陽に指定することなく簡潔に記述できるようになる.

2.3.1 正方形詰込み問題

一つめの例は正方形詰込み問題を解くプログラムである. 正方形詰込み問題 $SP(n, s)$ は一辺の長さ 1 から n まで 1 ずつ増加する正方形の集合を一辺の長さ s の正方形の枠内に重なりなく配置する問題である. 最も素直なモデリングは整数変数 $x_i, y_i \in \{0, \ldots, s-i\}$ をそれぞれの正方形 $i (1 \leq i \leq n)$ に $(x_i,\ y_i)$ が正方形 i の左下の座標を指すようにするものである. 以下の制約は任意の二つの正方形 i と j (但し $1 \leq i < j \leq n$) が重なることを禁止する.

$$(x_i + i \leq x_j) \quad \vee \quad (x_j + j \leq x_i) \ \vee$$
$$(y_i + i \leq y_j) \quad \vee \quad (y_j + j \leq y_i)$$

図 4 に $SP(15, 36)$ の時の Scarab プログラムを示す. Scarab DSL を用いることで簡潔に上記のモデルを表現できている. 4, 5 行目の `int` メソッドは整数変数をデフォルト CSP に定義している. ここで `'x`, `'y` は Scala におけるシンボルオブジェクトを表しており, 前述のように Scala の暗黙変換により整数変数オブジェクトへと変換される. 9 から 12 行目の `add` メソッドは上述の制約式を定義している. 14 行目の `find` メソッドは, 定義された CSP を SAT へと符号化した後に解を計算, 逆符号化している. 逆符号化された解は `solution` によって返され, Scala の `println` メソッドにより出力される.

```
 1:  val n: Int = 5
 2:  val pos = (0 until n)
 3:
 4:  for (i <- pos; j <- pos)
 5:    int('x(i,j),1,n)
 6:  for (i <- pos) {
 7:    add(alldiff(pos.map(j => 'x(i,j))))
 8:    add(alldiff(pos.map(j => 'x(j,i))))
 9:    add(alldiff(pos.map(j => 'x(j,(i+j+n-1)%n))))
10:    add(alldiff(pos.map(j => 'x(j,(i-j+n-1)%n))))
11:  }
12:
13:  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

図 5　$PLS(5)$ の Scarab プログラムと計算された解

2.3.2　汎対角線ラテン方陣

もう一つの例は CSP ソルバー競技会 [18] で使用された汎対角線ラテン方陣 (Pandiagonal Latin Square) である．汎対角線ラテン方陣 $PLS(n)$ は n 行 n 列の行列に 1 から n までの n 個の異なる整数を，各整数が各行，各列，各汎対角線に 1 回だけ現れるように配置する問題である．いま $PLS(n)$ が与えられたとき，整数変数を要素にもつ n 行 n 列の行列 $x_{i,j} \in \{1,\ldots,n\}$ $(1 \leq i,j \leq n)$ を用いてモデリングを行う．各整数が 1 回だけ現れる制約は *alldiff* 制約 [26] によって表す．この制約は制約プログラミングの分野で最もよく知られているグローバル制約の一つであり，与えられた n 個の整数変数が互いに異なることを意味する [37].

図 5 は $PLS(5)$ に対する Scarab プログラムを表している．Scarab DSL は *alldiff* を用いたモデリングを Scala の特長を用いて簡潔に表している．例えば 7 行目の `alldiff(pos.map(j => 'x(i,j)))` は各行においてそれぞれの変数が異なることを表す制約 $alldiff(x_{i,1}, x_{i,2}, \ldots, x_{i,n})$ を表している．

3　Scarab における SAT 符号化

ここまで Scarab プログラムについて説明した．ここからは Scarab プログラムで記述された制約の SAT 符号化について説明する．

Scarab では与えられた CSP を，`Simplifiler` クラス (4 節の図 8 参照) の `simplify` メソッドを用いて前処理する．これによりまず連言標準形の CSP に変換している．この連言標準形の CSP におけるリテラルは，ブール変数，ブール変数の否定，$\sum_{i=1}^{n} a_i x_i \leq c$ の形の線形制約のいずれかである．ここで a_i は非零の整数, c は整数値, そして x_i は整数変数を表す．なお連言標準形の CSP への変換は Tseitin 変換 [36] [32] を用いて新しいブール変数を導入する方法で行っている．

以下では Scarab で採用している SAT 符号化である順序符号化とその実装について説明する．

3.1　順序符号化

順序符号化法 [30] ではブール変数 $p(x \leq c)$ がそれぞれの整数変数 x と整数値 $c \in \{lb(x)-1,\ldots,ub(x)\}$ に対して導入される．ここで $lb(x)$ と $ub(x)$ はそれぞれ x の下限値と上限値である．さらに整数変数における値の順序関係を表す以下の制約が導入される．

$$\bigwedge_{c=lb(x)+1}^{ub(x)-1} (\neg p(x \leq c-1) \vee p(x \leq c))$$

順序符号化法では線形制約 $\sum_{i=1}^{n} a_i x_i \leq c$ を符号化するために図 6 に示す変換式を用いる．この変換を再帰的に適用し，変換式 (1a) と (1b) の $x_1 \leq \lfloor c/a_1 \rfloor$ と $x_1 \leq \lceil c/a_1 \rceil - 1$ をそれぞれブール変数 $p(x_1 \leq \lfloor c/a_1 \rfloor)$ と $p(x_1 \leq \lceil c/a_1 \rceil - 1)$ に変換することで最終的に線形制約はブール変数上の連言標準形式へと変換される．

$$\sum_{i=1}^{n} a_i x_i \le c = \begin{cases} x_1 \le \lfloor c/a_1 \rfloor & (n=1, a_1 > 0) \quad (1a) \\ \neg\left(x_1 \le \lceil c/a_1 \rceil - 1\right) & (n=1, a_1 < 0) \quad (1b) \\ \bigwedge_{b=lb(x_1)}^{ub(x_1)} \left((x_1 \le b-1) \vee \sum_{i=2}^{n} a_i x_i \le c - a_1 b \right) & (n \ge 2, a_1 > 0) \quad (1c) \\ \bigwedge_{b=lb(x_1)}^{ub(x_1)} \left(\neg(x_1 \le b) \vee \sum_{i=2}^{n} a_i x_i \le c - a_1 b \right) & (n \ge 2, a_1 < 0) \quad (1d) \end{cases}$$

図 6　順序符号化における変換

```
 1:  def encodeLe(axs: Seq[(Int,Var)], c: Int):
 2:         Seq[Seq[Literal]] = axs match {
 3:
 4:    case Seq((a,x)) =>
 5:      if (a > 0) Seq(Seq(le(x, floorDiv(c, a))))
 6:      else Seq(Seq(le(x, ceilDiv(c, a)-1).neg))
 7:    case Seq((a,x), axs1 @ _*) => {
 8:      if (a > 0) {
 9:        for {
10:          b <- lb(x) to ub(x)
11:          clause <- encodeLe(axs1, c-a*b)
12:        } yield le(x, b-1) +: clause
13:      } else {
14:        for {
15:          b <- lb(x) to ub(x)
16:          clause <- encodeLe(axs1, c-a*b)
17:        } yield le(x, b).neg +: clause
18:      }
19:    }
20:  }
```

図 7　順序符号化の実装

3.2　順序符号化の実装

図 7 は図 6 の変換を実装したプログラムを示している．1 行目は二つの入力を表しており，$\{(a_1, x_1), \ldots, (a_n, x_n)\}$ と整数値 c である．5，6 行目はそれぞれ変換式 (1a) と (1b) を表している．9 から 12 行目は変換式 (1c) を表している．11 行目では $\{(a_2, x_2), \ldots, (a_n, x_n)\}$ と $c - a_1 b$ を入力として `encodeLe` を再帰的に呼び出している．14 から 17 行目は同様に変換式 (1d) を表している．

例として正方形詰込み問題 $SP(2, 6)$ の制約の一部である制約 $x_1 + 1 \le x_2$ を考える．ここで $x_1 \in \{0, \ldots, 5\}$ と $x_2 \in \{0, \ldots, 4\}$ とする．まず初めに $x_1 + 1 \le x_2$ は線形制約 $x_1 - x_2 \le -1$ に変形される．図 7 の `encodeLe` を入力 $\{(1, x_1), (-1, x_2)\}$ と -1 に入力として呼ぶことにより，以下の順序符号化された CNF 式が得られる:

$$\begin{gathered} \neg p(x_2 \le 0) \ \wedge \\ (p(x_1 \le 0) \vee \neg p(x_2 \le 1)) \ \wedge \\ (p(x_1 \le 1) \vee \neg p(x_2 \le 2)) \ \wedge \\ (p(x_1 \le 2) \vee \neg p(x_2 \le 3)) \ \wedge \\ p(x_1 \le 3) \ \wedge \ p(x_1 \le 4) \end{gathered}$$

図 7 のように Scarab 中の順序符号化の中心部分の実装は 20 行程度と非常にコンパクトであり，SAT 型システム開発者が SAT 符号化を変更し易いようになっている．図 8 のクラス図において SAT 符号化の仕様は `Encoder` という抽象クラスに記述されており，順序符号化は `OrderEncoder` に実装されている．この `Encoder` クラスを実装することでこれまで提案されてきた他の SAT 符号化を実装することも可能である [14] [39] [11] [10] [1] [9] [35].

4　SAT ソルバー API と高度な解法

ここまで Scarab における制約記述，制約の SAT 符号化について 2 節，3 節で説明した．本節では SAT 符号化された結果得られた SAT 問題を解くために用いる抽象クラス `SatSolver` (図 8) についてまず説明を行う．この API を通じた SAT ソルバーとの密な連携は Scarab の大きな特長の一つとなっている．その後インクリメンタル解法，仮説を用いた解法，制約のコミットとロールバック，最適値の探索，組込み制約などの高度な SAT 技術の利用について説明を行う．

4.1　SAT ソルバー API

図 9 に SAT ソルバー API の一部を記述する．

2 行目から 5 行目までは SAT ソルバーに制約を追加するメソッドになる．`addClause` は最も基本的なもので SAT 問題における節を SAT ソルバーへと追加する．`addAtLeast`, `addAtMost`, `addExactly` は基数制約を (SAT 符号化せずに) そのまま SAT ソル

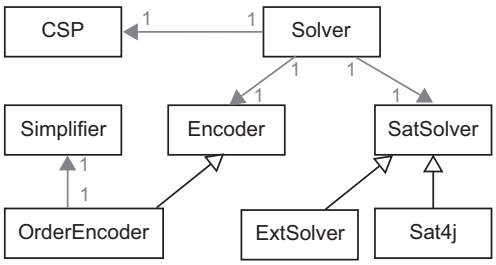

図 8　Scarab におけるソルバークラス図 (一部)

```
 1: abstract class SatSolver {
 2:   def addClause(lits: Seq[Int])
 3:   def addAtLeast(lits: Seq[Int], degree: Int)
 4:   def addAtMost(lits: Seq[Int], degree: Int)
 5:   def addExactly(lits: Seq[Int], degree: Int)
 6:   def addConstr(c: org.sat4j.specs.Constr)
 7:   def isSatisfiable: Boolean
 8:   def isSatisfiable(assumps: Seq[Int]): Boolean
 9:   def model: Array[Int]
10:   def clearLearntClauses
11:   def reset
      ... }
```

図 9　SAT ソルバー API (一部)

バーに追加する．これらのメソッドを実装する SAT ソルバーでは組込みの基数制約を扱える必要がある．`addConstr` はユーザが定義した任意の組込み制約を SAT ソルバーへと追加するメソッドである．組込み基数制約, 組込み制約の応用については 4.6 節と 4.7 節で説明する．

7, 8 行目は解計算を行うメソッドである．`isSatisfiable` は SAT ソルバーがメソッド実行時点で持っている制約を充足する解の存在を判定する．`isSatisfiable(assumps: Seq[Int])` も基本的な動作は同じであるがリテラルの連言を仮説として SAT ソルバーに与える点が異なる．SAT ソルバーにおける仮説の扱いについては文献 [22] [7] が詳しい．9 行目の `model` は問題が充足可能であることが `isSatisfiable` で判定された後に呼ぶことで充足解を返す．10 行目の `clearLearntClauses` は現時点で保有している学習節を削除するメソッド，11 行目の `reset` はソルバーの節や制約などの内部状態を初期化するメソッドである．

現時点の Scarab では `Sat4j` クラスと `ExtSolver` クラスが抽象クラス `SatSolver` を実装している (図 8 参照)．`Sat4j` クラスは Java で実装された SAT ソルバー Sat4j を用いて全てのメソッドを実装する．Scarab と同一の JVM 上で Sat4j のインスタンスが作成され全ての連携はメモリ上で行われる．`ExtSolver` は外部 SAT ソルバー実行のためのクラスであり一部の基本的なメソッドのみを実装する．`addClause` が呼ばれると問題ファイルに節の書き込みを行い，`isSatisfiable` が呼ばれると SAT ソルバーが外部プロセスとして起動され，問題ファイルの読み込みと求解を行う．C や C++で実装された最新の SAT ソルバーを利用できることが利点であるが，Scarab との連携がファイル読み書きを通じて行われるためオーバヘッドが生じる．このことについては Java Native Interface (JNI) や Java Native Access (JNA) を使うことで解決できるが，別途導入のコストがかかることになる．また最新の SAT ソルバーの多くは節以外の制約を利用できるような設計になっておらず，組込み制約に関する機能は制限される．以降は `SatSolver` の全てのメソッドを実装可能な SAT ソルバー Sat4j の使用を仮定して説明を行う．

4.2　インクリメンタル解法

前節では `SatSolver` クラスについて説明した．本節では特にインクリメンタル解法について Scarab DSL からそれらが実際どのように利用出来るかを説明する．

図 10 の 6〜9 行目に記載されるように, Scarab では SAT ソルバーが一度解を計算した後でも制約の追加が可能である．7 行目の 1 回目の `find` メソッドでは定義された CSP 全体が SAT 符号化され，生成された節集合が Sat4j へと追加される．次に SAT ソルバー Sat4j が求解を行う．9 行目の 2 回目の `find` メソッドでは 8 行目で追加された制約 $x \neq 3$ のみが符号化され，節集合が Sat4j に追加される．そして Sat4j が再び求解を行う．

ここで 1 回目の `find` メソッドによって生成された学習節は 2 回目の呼び出し時にも保持されており，学習節の再利用による効果的な解探索を期待できる (インクリメンタル解法)．但し，1 回目の CSP が充足不能の場合には制約の追加は許可されないので注意さ

```
 1:  // CSP 定義
 2:  int('x, 1, 3) // x ∈ {1,2,3}
 3:  int('y, 1, 3) // y ∈ {1,2,3}
 4:  add(x === y)  // 制約の定義 x = y
 5:
 6:  // 4.1 インクリメンタル解法
 7:  find          // 充足可能: x = 3, y = 3
 8:  add(x !== 3)  // 制約の追加
 9:  find          // 充足可能: x = 2, y = 2
10:
11:  // 4.2 Assumption を用いた解法
12:  find(y === 3) // 充足不能
13:  find(x === 1) // 充足可能: x = 1, y = 1
14:
15:  // 4.3 コミットとロールバック
16:  commit        // コミットポイント生成
17:  add(x < y)    // x < y が追加される
18:  find          // 充足不能
19:  rollback      // x < y の追加をロールバック
20:  find          // 充足可能: x = 2, y = 2
```

図 10　Sat4j を用いた高度な解法の例

れたい．

4.3　仮説を用いた解法

Scarab DSL では制約を引数に持つ `find(assump: Constraint)` メソッドを用いることで CSP レベルで仮説 (Assumption) に基づく解探索が可能である．但し，現状は仮説に指定する制約はブール変数上のリテラルの連言に符号化される必要があり，それ以外の制約が指定された場合には例外が発生する．それぞれのリテラルは Sat4j へと渡されこの仮説リテラル集合に基づく SAT の解探索が Sat4j で行われる．

図 10 の例では，12 行目で仮説 $y = 3$ のもとで CSP の求解が行われている．しかし，この仮説はこれまで追加された制約 $(x = y) \wedge (x \neq 3)$ と矛盾するので充足不能が返される．続いて仮説 $x = 1$ のもとで CSP の求解が行われ，この場合には充足可能となる．

このように仮説で指定した制約は CSP に永続的に追加されるのではなく，求解中にのみ有効であり，また探索中に得られた学習節も保持される．後述するように，この仮説を用いた解法は最適解のデクリメンタル探索や二分法へと応用することができる．

4.4　制約のコミットとロールバック

Scarab では以下のような制約のコミット (`commit`) とロールバック (`rollback`) メソッドを提供している．

- `commit` メソッドは現在の CSP の状態 (整数変数，ブール変数，制約の数) を記録する．現在の Scarab の実装では一つのコミットポイントを作成可能である．
- `rollback` メソッドは CSP を最後のコミットポイントの状態まで戻す．同時に Sat4j の状態も `reset` メソッドにより初期化されるので次回の `find` メソッド呼び出し時には Sat4j へと節を追加し直す必要がある．

これら `commit`/`rollback` メソッドを実行することで，これまでに追加した制約の削除が可能となり，動的な制約の変更が必要なアプリケーションに対して，より柔軟な解探索を提供することができると考えられる．しかし，符号化の手続きは繰り返し行う必要があり，CSP が充足不能になった場合には学習節は再利用できないので注意が必要である．

図 10 の例では，16 行目でコミットポイントが生成され，17 行目で制約 $x < y$ が追加されている．しかし，この制約はこれまで追加された制約 $(x = y)$ と矛盾するので充足不能が返される．次に 19 行目で `rollback` メソッドが呼ばれ CSP が 16 行目の状態まで戻される．すなわち制約 $(x = y)$ が削除される．20 行目で再び求解が行われ，この場合には CSP は充足可能となる．

4.5　最適値の探索

本節では上述の 3 つの機能を用いた応用として解の最適化を紹介する．例として，図 4 に示した正方形詰込み問題 $SP(n, s)$ について，制約を充足するような最小の正方形の枠の大きさを計算する．

図 11 は仮説を用いたデクリメンタル探索のプログラムを示している．このプログラムは図 4 のプログラムの最下部に追加することで実行可能であり，プログラム中の Scala の整数変数 `n` と `s` は図 4 で定義されているものである．

まずプログラム 3 行目の整数変数 $m \in \{lb, \ldots, ub\}$ は正方形の枠のサイズを表している．5，6 行目で定義される制約は全ての正方形がサイズ m の枠をはみ

```
 1:  val lb = n
 2:  var ub = s
 3:  int('m, lb, ub)
 4:
 5:  for (i <- 1 to n)
 6:    add(('x(i)+i <= 'm) && ('y(i)+i <= 'm))
 7:
 8:  while (lb <= ub && find('m <= ub)) {
 9:    add('m <= ub)
10:    ub = solution.intMap('m) - 1
11:  }
12:
13:  while (findNext)
14:       println(solution)
```

図 11　仮説を用いたデクリメンタル探索

出ないことを保証する．8 行目では find メソッドが仮説 $m \leq ub$ とともに呼ばれている．もし解が存在するならば, 制約 $m \leq ub$ が追加され，ub が m の最も最近の値から 1 つ小さい値に更新される (10 行目)．13，14 行目では全ての最適解が列挙されている．findNext は最後に得られた解の否定をブロック節として加えることで別の解の探索を行うようになっている．

図 12 は commit/rollback メソッドと制約の追加を用いた二分探索を示している．6 行目は現在の下限と上限のちょうど半分の値を計算している．7，8 行目は正方形の枠の大きさを制限する制約を追加している．9 行目では, find メソッドが呼ばれている．もし CSP が充足可能である場合, 上限が更新され現在の CSP がコミットされる (10，11 行目)．もし充足不能である場合, 下限が 1 だけ増加され最後のコミットポイントへと CSP がロールバックされる．

4.6　組込み基数制約による符号化クラスの拡張

本節では 4.1 節で説明した SatSolver クラスの中で特に addAtLeast, addAtMost, addExactly を用いた SAT 符号化クラス OrderEncoder の拡張について説明する．

ブール基数制約 (以後，基数制約) とは，0-1 変数の値が 1 になるものに上限または下限を設定する制約であり，$\sum_{i-1}^{n} x_i \triangleright k$ の形式のものをいう．但し，ここで i は整数変数, n は変数の数, k はしきい値を表す整数とする．比較演算子 $\triangleright$ には $\leq$, $\geq$, $=$ のいずれかが入る．基数制約で表現できる組合せ問題は非常に

```
 1:  var lb = n
 2:  var ub = s
 3:  commit
 4:
 5:  while (lb < ub) {
 6:    var size = (lb + ub) / 2
 7:    for (i <- 1 to n)
 8:      add(('x(i)+i<=size)&&('y(i)+i<=size))
 9:    if (find) {
10:      ub = size
11:      commit
12:    } else {
13:      lb = size + 1
14:      rollback
15:    }
16:  }
```

図 12　commit/rollback メソッドを用いた二分探索

多く，CSP ソルバーにとっても重要な制約である．

通常 SAT ソルバーはリテラルの選言である節の集合を入力として受け取るため，SAT ソルバーを用いて解を求めるためには基数制約を節の集合に符号化する必要がある．基数制約の SAT 符号化は活発な研究分野の一つでありこれまで次のような方法が提案されてきた (括弧内は符号化節数): Binomial (${}_nC_{k+1}$), Totalizer [2] ($O(n \cdot k)$), Sequential Counter [27] ($O(n \cdot k)$). Binomial と比較して Toalizer と Sequential Counter は符号化節数が改良されているが，n と k が大きくなるとやはり多くの節が出力される．

Sat4j は他の多くの SAT ソルバーと異なり，節以外の制約を Conflict Driven Clause Learning (CDCL) の探索で扱うことが可能である．CDCL の詳細については [22] に詳しい．本論文ではこのように SAT ソルバー内で扱うことができる節以外の制約を**組込み制約**と呼ぶ．Sat4j はこの組込み制約のためのインターフェース Constr が用意されており，**組込み基数制約**は標準で実装されている組込み制約の一つである．組込み基数制約は 4.1 節に記載した addAtLeast, addAtMost, addExactly を使って Scarab から Sat4j に入力される．

Scarab ではこの組込み基数制約を使った SAT 符号化クラス NativeEncoder を実装している．NativeEncoder は 3 節で説明した SAT 符号化クラス OrderEncoder を継承しており, 基本的に順序符

表 1　汎対角線ラテン方陣における性能評価 (CPU 時間: 秒)

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

号化を行うが基数制約を検知すると節へと符号化せずに `addAtLeast`, `addAtMost`, `addExactly` メソッドを使って Sat4j に直接組込み基数制約を入力する．

`NativeEncoder` は定義された CSP に基数制約が多く含まれている時に SAT ソルバーに追加される節の増加を抑えるため，符号化時間を短縮することと SAT ソルバーにおける単位伝播効率の悪化を防ぐことを期待出来る．`NativeEncoder` を使った場合の性能評価については 5.2 節に記載する．

4.7　ユーザによる組込み制約の定義

前節では Sat4j における組込み基数制約とそれを用いた新しい SAT 符号化クラスについて説明を行った．ここではさらなる拡張として，組込み制約自体をユーザが定義する方法について説明する．

`SatSolver` クラスは `addConstr` メソッドを備えており，これは Sat4j における `org.sat4j.specs.Constr` を実装した組込み制約を SAT ソルバーに追加する．すなわち既に定義された基数制約だけでなく，ユーザが定義さえすれば任意の制約を SAT ソルバーで利用できるようになる．

例えば図 14 は基数制約に係数 a_i をつけた擬似ブール制約 $\sum_{i-1}^{n} a_i x_i \triangleright k$ を組込み制約として Scala 上で定義したプログラムを表している．一行目で Sat4j の `org.sat4j.specs.Constr` を拡張して定義していることに注意されたい．組込み制約を定義するためには大きく分けて `register` (10〜19 行目), `propagate` (21〜31 行目), `calcReason` (33 行目〜40 行目) の 3 つのメソッドを定義するだけで良い．簡単に説明すると `register` には制約が SAT ソルバーに追加された時の動作を記述する．主に監視リテラルの設定になる．`propagate` は監視中のリテラルの充足・非充足を基に値の伝播を行う．`calcReason` は矛盾が発生した際に原因となった真偽値割当てをリテラルの連言として生成する．最終的にこのように定義した制約は `addConstr` によって SAT ソルバーに追加することができる．

近年，論文 [6] のような節でない制約を SAT ソルバーに取り入れる研究がされているが，Scarab を用いることでこのような高度な SAT 技術の開発コストを少なくすることを期待できる．

5　性能評価

5.1　汎対角線ラテン方陣

Scarab の基本性能を評価するために図 5 で示した汎対角線ラテン方陣を用いて計算機実験を行った．ここでは二つの *alldiff* 制約の実装を用いた．一つめは `naive` 版で *alldiff* 制約の定義に従い与えられた n 個の変数 $x_1, \ldots, x_n$ の入力に対し，$\bigwedge_{1 \le i < j \le n} (x_i \neq x_j)$ なる制約を定義するものである．二つめは `optimized` 版で，順列制約 $\bigwedge_{i=lb}^{ub} \bigvee_{j=1}^{n} (x_j = i)$ と鳩ノ巣原理の制約 $\neg \bigwedge (x_i < lb + n - 1)$ and $\neg \bigwedge (x_i > ub - n + 1)$ を `naive` 版に加えたものである．この鳩の巣原理の制約の効果は文献 [29] で報告されている．なお `naive` 版は Scarab では 2 行で実装されており `optimized` 版でも 15 行程度である．このように制約の実装を簡潔に記述でき，改良したものを実験できることは Scarab で SAT 型システム開発を行う利点の一つである．

変換制限時間は 1 時間で，全ての計算時間は Xeon 2.93GHz の CPU，JVM に対して 2GB のメモリを割当てた MacOS X 上で行っている．

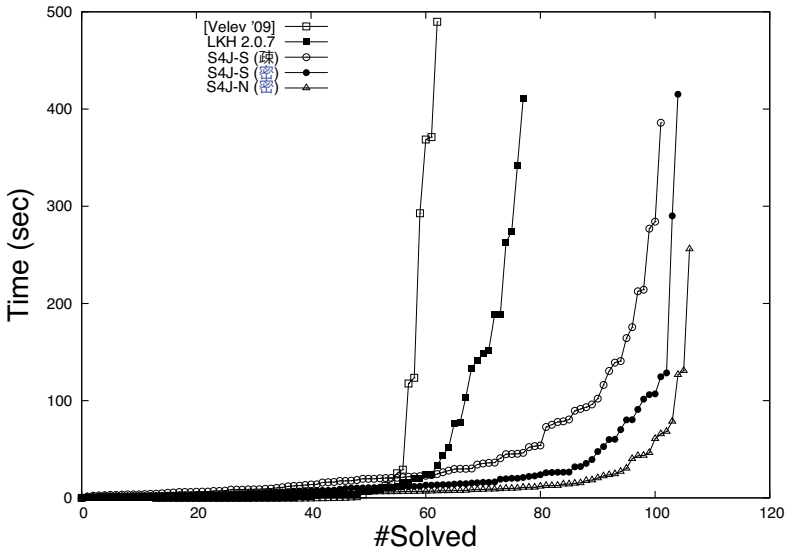

図 13　Scarab 上に実装された CEGAR と `NativeEncoder` を使った HCP の解法 [28]

表 1 は $PLS(n)$ (但し $3 \leq n \leq 16$) に対する Scarab の計算時間 (CPU 時間，秒) を示している．`optimized` 版 *alldiff* 制約を用いたものは $n = 16$ までを解くことに成功している．2009 年に開催された CSP ソルバー競技会では $n \leq 12$ までを解いた Sugar を除くと $n > 8$ の $PLS(n)$ を 1800 秒以内にどの CSP ソルバーも解くことができなかったことから Scarab はこの問題において良い性能を示しているといえる．

5.2　ハミルトン閉路問題

ハミルトン閉路問題 (HCP) は与えられたグラフの全頂点を通る閉路を求める NP 完全問題である．グラフ理論における重要な問題であり，計算機科学においても巡回セールスマン問題 (TSP) との関係があり重要である．本節では Scarab 上に実装された Counterexample Guided Abstraction Refinement (CEGAR) [5] と `NativeEncoder` および `OrderEncoder` を使った HCP の解法の性能を論文 [28] の結果を用いて説明する．

CEGAR を使って問題を解く場合，(i) 与えられた問題はまず緩和され, (ii) 求解を行ったうえで誤った解が出力されればそれを修正するように制約が追加されていく．SAT ソルバーは結果的に多くの回数起動する必要がある．本手法の詳細については論文 [28] とその補足ページ†3 を参照されたい．補足ページに記載があるが Scarab を使うことで本手法は問題の読み込みも含めて 100 行程度で記述することができる．比較に用いた手法は以下である．

Velev　全制約を SAT 符号化する従来手法 [38]

LKH　DIMACS TSP チャレンジの全記録を保持する HCP/TSP ソルバー LKH

S4J-N (密)　CEGAR + `NativeEncoder`

S4J-S (密)　CEGAR + Sequential Counter

S4J-S (疎)　S4J-S と同じ．しかし `ExtSolver` インターフェースで Sat4j を外部プロセスで起動して疎な連携を模擬．

ここで S4J-S (密) と S4J-S (疎) は両方とも `OrderEncoder` を用いて Sequential Counter を実装している．

図 13 は解いた問題数と CPU 時間 (カクタスプロット) で各手法を比較したものである．結果より S4J-N が最も右下にきており，最も高速に多くの問題を解いたことが分かる．HCP を CEGAR を用い

†3　http://kix.istc.kobe-u.ac.jp/~soh/scarab/jelia2014/

て解く場合には制約はほとんど基数制約になるため NativeEncoder を用いる S4J-N が OrderEncoder を用いた Sequential Counter を実装した S4J-S (密) よりも良い結果となった．また実装が異なる S4J-S (密) と S4J-S (疎) の実験結果の違いは SAT ソルバーを複数回起動する CEGAR メソッドを使った場合の起動コストとファイル読み書きによるオーバヘッドの影響が小さくないことを表しており，Scarab のような SAT ソルバーと密に連携するシステムで実装することの重要性を示している．全制約をまず最初に全て SAT 符号化する Velev と S4J-N の違いは顕著であり CEGAR のような高度な SAT 解法の必要性が読み取れる．また最新の TSP ソルバーである LKH と比較しても S4J-N は高速に多くの問題を解いている．

最後に上記で使われた問題から構造的な問題を多く含む `color04` (119 問) を用いて Sugar と C++で実装された Minisat を内部に持つ Copris で CEGAR と Sequential Counter を実装して解いたところ制限時間 500 秒以内に 84 問を解いた．S4J-N は同じ制限時間内に 86 問を解いており良い性能を示している．

6 おわりに

この論文では Scala 上に実装された SAT 型制約プログラミングシステム開発ツールである Scarab の説明を行った．正方形詰込み問題と汎対角線ラテン方陣の二つの例によって示されるように Scarab DSL と Scala の特長を利用することで SAT 型システム開発者は簡潔に問題のモデリングを行うことができる．

また Sat4j の機能を用いることで, Scarab は次の機能を提供する: インクリメンタル探索; 仮説を用いた CSP の解探索; 制約のコミットとロールバック; 組込み制約．これらの機能は Scarab において最適化，解列挙，CEGAR，新たな組込み制約などの高度な機能を実装するのに使うことができる．

Scarab の興味深い拡張の一つとして Sat4j の機能をさらに導入することがある．例えば Sat4j は与えられた CNF 式が充足不能である時に Minimal Unsatisfiable Subformula (MUS) を計算することが可能である．加えて埋め込みの最適化メソッドおよび解列挙メソッド，また基数制約や擬似ブール制約なども入力として扱うことが可能である．Scarab が持つ制約表現および SAT 符号化モジュールとこれら Sat4j の機能の組合せは SAT 技術のさらに応用を広げることを期待できる．Scarab の情報とソースコードは以下から入手できる．
`http://kix.istc.kobe-u.ac.jp/~soh/scarab/`.

参 考 文 献

[1] Ansótegui, C. and Manyà, F.: Mapping Problems with Finite-Domain Variables into Problems with Boolean Variables, *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT 2004), LNCS 3542*, 2004, pp. 1–15.

[2] Bailleux, O. and Boufkhad, Y.: Efficient CNF Encoding of Boolean Cardinality Constraints, *Proceedings of the 9th International Joint Conference on Principles and Practice of Constraint Programming (CP 2003), LNCS 2833*, 2003, pp. 108–122.

[3] 番原睦則, 田村直之: SAT によるシステム検証, 人工知能学会誌, Vol. 25, No. 1(2010).

[4] Biere, A., Heule, M., van Maaren, H., and Walsh, T.(eds.): *Handbook of Satisfiability*, Frontiers in Artificial Intelligence and Applications (FAIA), Vol. 185, IOS Press, 2009.

[5] Clarke, E. M., Grumberg, O., Jha, S., Lu, Y., and Veith, H.: Counterexample-Guided Abstraction Refinement, *CAV*, 2000, pp. 154–169.

[6] Dvorák, W., Järvisalo, M., Wallner, J. P., and Woltran, S.: Complexity-sensitive decision procedures for abstract argumentation, *Artif. Intell.*, Vol. 206(2014), pp. 53–78.

[7] Eén, N. and Sörensson, N.: Temporal Induction by Incremental SAT Solving, *Electronic Notes in Theoretical Computer Science*, Vol. 89, No. 4(2003).

[8] Ganesh, V., O'Donnell, C. W., Soos, M., Devadas, S., Rinard, M. C., and Solar-Lezama, A.: Lynx: A Programmatic SAT Solver for the RNA-Folding Problem, *SAT*, 2012, pp. 143–156.

[9] Gavanelli, M.: The Log-Support Encoding of CSP into SAT, *Proceedings of the 13th International Joint Conference on Principles and Practice of Constraint Programming (CP 2007), LNCS 4741*, 2007, pp. 815–822.

[10] Gent, I. P. and Nightingale, P.: A New Encoding of Alldifferent into SAT, *Proceedings of the 3rd International Workshop on Modelling and Reformulating Constraint Satisfaction Problems*, 2004.

[11] Gent, I. P.: Arc Consistency in SAT, *Proceedings of the 15th European Conference on Artificial Intelligence (ECAI 2002)*, 2002, pp. 121–125.

[12] Hebrard, E., O'Mahony, E., and O'Sullivan, B.: Constraint Programming and Combinatorial Optimisation in NumberJack, *Proceedings of CPAIOR*, 2010, pp. 181–185.

[13] 井上克巳, 田村直之: SAT ソルバーの基礎, 人工知

能学会誌, Vol. 25, No. 1(2010).
[14] Iwama, K. and Miyazaki, S.: SAT-Variable Complexity of Hard Combinatorial Problems, *Proceedings of the IFIP 13th World Computer Congress*, 1994, pp. 253–258.
[15] Janota, M., Grigore, R., and Marques-Silva, J.: Counterexample Guided Abstraction Refinement Algorithm for Propositional Circumscription, *JELIA*, 2010, pp. 195–207.
[16] Janota, M., Klieber, W., Marques-Silva, J., and Clarke, E. M.: Solving QBF with Counterexample Guided Refinement, *SAT*, 2012, pp. 114–128.
[17] Le Berre, D. and Parrain, A.: The Sat4j library, release 2.2, *Journal on Satisfiability, Boolean Modeling and Computation*, Vol. 7, No. 2-3(2010), pp. 59–6.
[18] Lecoutre, C., Roussel, O., and van Dongen, M. R. C.: Promoting Robust Black-box Solvers Through Competitions, *Constraints*, Vol. 15, No. 3(2010), pp. 317–326.
[19] Mernik, M., Heering, J., and Sloane, A. M.: When and how to develop domain-specific languages, *ACM Comput. Surv.*, Vol. 37, No. 4(2005), pp. 316–344.
[20] Metodi, A. and Codish, M.: Compiling finite domain constraints to SAT with BEE, *TPLP*, Vol. 12, No. 4-5(2012), pp. 465–483.
[21] 鍋島英知: SAT によるプランニングとスケジューリング, 人工知能学会誌, Vol. 25, No. 1(2010).
[22] 鍋島英知, 宋剛秀: 高速 SAT ソルバーの原理, 人工知能学会誌, Vol. 25, No. 1(2010).
[23] 則武治樹, 番原睦則, 宋剛秀, 田村直之, 井上克巳: パッキング配列問題の制約モデリングと SAT 符号化, コンピュータソフトウェア, Vol. 31, No. 1(2013), pp. 116–130.
[24] Odersky, M.: The Scala Experiment — Can We Provide Better Language Support for Component Systems?, 2006.
[25] Odersky, M., Spoon, L., and Venners, B.: *Programming in Scala*, Artima, Inc., second edition, 2010.
[26] Régin, J.-C.: A Filtering Algorithm for Constraints of Difference in CSPs, *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI 1994)*, 1994, pp. 362–367.
[27] Sinz, C.: Towards an Optimal CNF Encoding of Boolean Cardinality Constraints, *Proceedings of the 11th International Joint Conference on Principles and Practice of Constraint Programming (CP 2005), LNCS 3709*, 2005, pp. 827–831.
[28] Soh, T., Le Berre, D., Roussel, S., Banbara, M., and Tamura, N.: Incremental SAT-based Method with Native Boolean Cardinality Handling for the Hamiltonian Cycle Problem, *Proceedings of the 14th European Conference on Logics in Artificial Intelligence*, to appear, 2014.
[29] 田島宏史: SAT 変換に基づく制約ソルバーの高速化に関する研究, 修士論文, 神戸大学大学院自然科学研究科, 2006.
[30] Tamura, N., Taga, A., Kitagawa, S., and Banbara, M.: Compiling Finite Linear CSP into SAT, *Constraints*, Vol. 14, No. 2(2009), pp. 254–272.
[31] Tamura, N., Tanjo, T., and Banbara, M.: System Description of a SAT-based CSP Solver Sugar, *Proceedings of the 3rd International CSP Solver Competition*, 2008, pp. 71–75.
[32] 田村直之, 丹生智也, 番原睦則: SAT 変換に基づく制約ソルバーとその性能評価, コンピュータソフトウェア, Vol. 27, No. 4(2010), pp. 183–196.
[33] 田村直之, 丹生智也, 番原睦則: 制約最適化問題と SAT 符号化, 人工知能学会誌, Vol. 25, No. 1(2010).
[34] 田村直之, 丹生智也, 番原睦則: Scala 上の制約プログラミング用ドメイン特化言語 Copris について, コンピュータソフトウェア, Vol. 29, No. 4(2012), pp. 114–129.
[35] 丹生智也, 田村直之, 番原睦則: 位取り記数法に基づく整数有限領域上の制約充足問題のコンパクトかつ効率的な SAT 符号化, コンピュータソフトウェア, Vol. 30, No. 1(2013), pp. 211–230.
[36] Tseitin, G. S.: On the complexity of derivations in the propositional calculus, *Studies in Mathematics and Mathematical Logic Part II*, (1968), pp. 115—–125.
[37] van Hoeve, W.-J. and Katrie, I.: Global Constraint, *Handbook of Constraint Programming*, Foundations of Artificial Intelligence, Elsevier, 2006, pp. 169–208.
[38] Velev, M. N. and Gao, P.: Efficient SAT Techniques for Relative Encoding of Permutations with Constraints, *Australasian Conference on Artificial Intelligence*, 2009, pp. 517–527.
[39] Walsh, T.: SAT v CSP, *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming (CP 2000)*, 2000, pp. 441–456.
[40] Zhou, N.-F.: The language features and architecture of B-Prolog, *Theory and Practice of Logic Programming*, Vol. 12, No. 1-2(2012), pp. 189–218.

```
 1: class NativePB(voc: ILits, ps: IVecInt, coef: Seq[Int], degree: Int)
                      extends Constr with Propagatable with Undoable {
 2:   val maxUnsatisfied = coef.sum - degree
 3:   var counter = 0
 4:   var remains: Set[Int] = Set.empty
 5:   var coefMap: Map[Int, Int] = Map.empty
 6:   val lits = new Array[Int](ps.size)
 7:   ps.moveTo(this.lits)
 8:   register
 9:
10:   def register {
11:     for (i <- 0 until lits.size) {
12:       voc.watch(lits(i) ^ 1, this)
13:       coefMap = coefMap + ((lits(i) >> 1) -> coef(i))
14:       if (voc.isFalsified(lits(i))) {
15:         voc.undos(lits(i) ^ 1).push(this)
16:         counter += coef(i)
17:       }
18:     }
19:   }
20:
21:   def propagate(s: UnitPropagationListener, p: Int): Boolean = {
22:     voc.watch(p, this)
23:     if ((counter + coefMap(p >> 1)) > maxUnsatisfied) return false
24:     counter += coefMap(p >> 1)
25:     voc.undos(p).push(this)
26:
27:     for (q <- lits if voc.isUnassigned(q) && ((counter + coefMap(q >> 1)) > maxUnsatisfied))
28:       if (!s.enqueue(q, this))
29:           return false
30:     true
31:   }
32:
33:   def calcReason(p: Int, outReason: IVecInt) {
34:     var c = 0
35:     for (q <- lits if voc.isFalsified(q)) {
36:       outReason.push(q ^ 1)
37:       c += coefMap(q >> 1)
38:       if (c > maxUnsatisfied) return
39:     }
40:   }
```

図 14　組込み擬似ブール制約の定義 (一部)