Nondeterministic Programming in Java with JSetL

Gianfranco Rossi, Federico Bergenti
Dipartimento di Matematica e Informatica
Università di Parma
General aim: providing facilities to support **nondeterministic programming** within **conventional** O-O programming languages (e.g., Java)

Constructs for expressing nondeterministic control are often present in high-level **specification/modeling** languages (e.g., Alloy, TLA+ language, Coq, Isabelle, ...)

In contrast, rarely taken into account in executable languages (only sequential control)

An exception, **Logic Programming** and its extensions (CLP, Functional Logic (e.g., Curry), multiparadigm (e.g., Oz-Mozart))

In imperative languages?
Context - Nondeterminism in imperative programming lang's

Very few proposals ("don't know" nondeterminism)

*SETL* [Schwartz et al., 1986]
*Kaleidoscope* [Lopez et al., 1994]
*Alma-0* [Apt, Schaerf, et al.]
*Turtle* [Grabmüller, Hofstedt, 2003]
Scala + Kaplan [Kuncak et al., 2012]

**New** languages – syntactic/semantics extensions
**Not mainstream** languages

--> difficult to be developed

--> difficult to be accepted and used by (rather conservative) programmers
Our proposal

Stay within conventional well-assessed O-O programming languages (namely, Java)

Using a purely **library-based approach** (no extensions)

Key notions (related to non-determinism)

**Sets**: set data structures and operations are inherently nondeterministic (e.g., \(x \in \{1, 2, 3\}\))

**Logical variables**: possibly unbound, not-modifiable though side-effects

**Constraints**: relations vs functions (more solutions), exploring the search-space

\[\Rightarrow\] using the **JSetL** library
**JSetL**

JSetL is a Java library that combines the object-oriented programming paradigm of Java with valuable concepts of CLP languages, such as logical variables, lists, unification, constraint solving, nondeterminism.

The library provides also sets and set constraints like those found in {log}. Unification may involve logical variables, as well as list and set objects ("set unification"). Constraints concern basic set-theoretical operations (e.g., membership, union, intersection, etc.), as well as equality, inequality and integer comparison operations.

Equality, inequality and comparison constraints on integers are dealt with as **Finite-Domain Constraints**.

Set constraints are solved using a complete solver that accounts for partially specified sets (i.e., sets containing unknown elements).
JSetL has been developed at the Department of Mathematics of the University of Parma (Italy).

It is completely written in **Java**. The full Java code of the JSetL library, along with sample programs and related documents, is available at [cmt.math.unipr.it/jsetl.html](http://cmt.math.unipr.it/jsetl.html).

The library is **free software**; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License.

---


Logical objects

Logical variables
Ex. - creation

```java
LVar x = new LVar("x");
IntLVar j = new IntLVar("j",1,10);
```

Ex. - operations

```java
x.isBound(); --&gt; true/false
x.output(); --> x = value or x unbound
 +
  constraints (e.g., x.eq("abc"), j.gt(0))
```

Logical sets/lists
Ex. - set creation

```java
LSet r = new LSet();   // the empty set
LSet s = new LSet().ins(1).ins(2);   // {1,2}
LSet t = r.ins(x).ins(1);   // {1,x|r}
```
Logical objects – cont’d

Ex. - list creation

```java
LList l = new LList().ins(1).ins(x); // [x,1]
```

Ex. - operations

```java
s.isBound(); --> true/false
s.getSize(); --> the cardinality of s
+ set constraints
  x.in(s), x.nin(s)
  t.union(r,s), t.inters(r,s),...
```

Note that constraints are objects

Ex.

```java
Constraint c = new Constraint("in",x,s);
Constraint cc = c.and(s.neq(LSet.empty()));
// x ∈ s & s ≠ { }
```
**Constraint solving**

Constraints are solved by the JSetL constraint solver = an object of type SolverClass

Ex: 
```java
SolverClass solver = new SolverClass();
```

Constraints must be added to the constraint store (CS) and then solved by using solver's methods (solve, check ...)

Ex:
```java
solver.add(x.gt(0).and(x.neq(10))); // x > 0 & x ≠ 10
solver.solve();
```

(or: `solver.solve(x.gt(0).and(x.neq(10)));` )

Constraints in the CS are reduced to either false or to a simplified form that is assured to be satisfiable (like in \{log\})
Embedded nondeterminism

Constraint solving in JSetL is nondeterministic. In particular, constraint solving over partially specified sets may involve nondeterminism.

Ex.: unifying two sets \( \{1,2\} = \{x,y\} \) (set unification)

```java
LSet s = new LSet().ins(1).ins(2);
LSet r = new LSet().ins(x).ins(y);
solver.solve(s.eq(r));
x.output(); y.output();  --> x = 1, y = 2
solver.nextSolution();
x.output(); y.output();  --> x = 2, y = 1
```
Exploiting the embedded nondeterminism – an example

Ex. (Permutations) Given a set of integer numbers \( s \), print all permutations of \( s \).

```java
public static void allPermutations(LSet s) {
    int n = s.getSize(); // the cardinality of \( s \)
    LSet r = LSet.mkLSet(n); // \( r = \{x_1,x_2,\ldots,x_n\} \)
    solver.check(r.eq(s)); // \( r = s \)
    do {
        r.printElems(' ');
        System.out.println();
    } while (solver.nextSolution());
}
```

\( s = \{1,3,5\} \)

1 3 5   1 5 3   3 1 5   5 1 3   3 5 1   5 3 1
General nondeterministic control structures

Embedded nondeterminism is not enough to represent general nondeterministic control structures

Nondeterministic construct:

\[ \text{either } S_1 \text{ orelse } S_2 \ldots \text{ orelse } S_n \text{ end} \]

Creates \textit{choice points}, that are explored through \textit{backtracking} the computation may terminate \textit{with success} or \textit{with failure}

A very simple example:

\begin{verbatim}
either
  x = 1;
orelse
  x = 2;
orelse
  x = 3;
end;
x > 1;
print(x);
\end{verbatim}
Our solution

Let $m$ be a method that uses nondeterminism

Define $m$ as a **new constraint**
by exploiting JSetL facilities for user-defined constraints

Within $m$, exploit JSetL facilities for creating and handling choice-points

Call $m$ as a constraint by using the solver;
the solver will process the constraint by using the embedded mechanisms for nondeterminism

User-defined constraints are defined as methods in a class that extends the library class `NewConstraintClass`
according to suitable programming conventions
Nondeterministic user-defined constraints

Ex.:

```java
public class NdTest extends NewConstraintsClass {
    ...
    public Constraint ndEq(LVar x) {
        return new Constraint("ndEq",x);
    }
    ...
    private void ndEq(Constraint c) {
        LVar x = (LVar)c.getArg(1);
        switch(c.getAlternative()) {
            case 0: Solver.addChoicePoint(c);
                    Solver.add(x.eq(1));
                    break;
            case 1: Solver.addChoicePoint(c);
                    Solver.add(x.eq(2));
                    break;
            case 2: Solver.add(x.eq(3));
                    break;
        }
    }
}
```

```plaintext
either
    x = 1;
orelse
    x = 2;
orelse
    x = 3;
end;

x > 1;
print(x);
```
Using the new constraint:

```java
LVar n = new LVar();
NdTest ndTest = new NdTest(solver);
solver.check(ndTest.ndEq(n).and(n.gt(1)));
System.out.println(n);   // --> 2
```

The remaining parts of classes that extend the class NewConstraintClass contain:

- the class constructor, that allows to set the solver that is assumed to manage these new constraints

- a “dispatcher” method, called user-code, that associates each new constraint name with the corresponding method; called by the specified solver
Nondeterministic user-defined constraints – cont’d

In order to implement nondeterministic procedures as user-defined constraints we must also take into account the following:

- using (reference to) **logical objects as arguments** provides a simple way for the method to return results (the method type is Constraint)

- using **logical objects as data** within constraint methods is fundamental to allow backtracking to restore the correct state when necessary

- logical objects need to be manipulated only through **constraints** (e.g., \( \text{eq} \) constraints instead of assignments)

```java
public class NdTest extends NewConstraintsClass {
  ...
  public Constraint ndEq(LVar x) {
    return new Constraint("ndEq",x);}
  ...
  private void ndEq(Constraint c) {
    LVar x = (LVar)c.getArg(1);
    switch(c.getAlternative()) {
      case 0: Solver.addChoicePoint(c);
               Solver.add(x.eq(1));
               break;
      case 1: Solver.addChoicePoint(c);
               Solver.add(x.eq(2));
               break;
      case 2: Solver.add(x.eq(3));
    }
  }
}
```
An example

Ex.: (Finding a path) Let $G$ be a directed graph and $s$ and $d$ two nodes of $G$. The problem consists in determining whether there is a path in $G$ from $s$ (the source node) to $d$ (the destination node).

In pseudo-code

```
path(G,s,d):
    either
        test([s,d] is an arc in G);
        return true;
    else
        test(there exists a node $t$ in $G$ such that $[s,t]$ is an arc in $G$ and path($G,t,d$));
        return true;
    end
    return false;
```

The new JSetL constraint

```
public Constraint path( LSet G,
    LVar s,
    LVar d,
    LList p)  //the computed path
```
private void path(Constraint c) {
LSet G = (LSet)c.getArg(1);   // the graph
LVar s = (LVar)c.getArg(2);   // the source node
LVar d = (LVar)c.getArg(3);   // the destination node
LList p = (LList)c.getArg(4); // the computed path
switch(c.getAlternative()) {
   case 0:
      solver.addChoicePoint(c);
      LList finalArc = LList.empty().ins(d).ins(s);   // [s,d]
      solver.add(G.contains(finalArc));              // finalArc ∈ G
      solver.add(p.eq(LList.empty().ins(finalArc))); // p=[[s,d]]
      break;
   case 1:
      LVar t = new LVar();                          // [s,t]
      LList intermediateArc = LList.empty().ins(t).ins(s));
      solver.add(G.contains(intermediateArc));      // intermediateArc ∈ G
      solver.add(path(G,t,d,t_dPath));
      solver.add(p.eq(t_dPath.ins(intermediateArc))); // p=[[s,t]|t_dPath]
   }
return;
}
Sample use

```java
    solver.solve(graphOps.path(G,s,d,p)
where  graphOps is an instance of the class containing path
```

Multiple uses of constraint methods

solving constraint $c(x,y)$ does not make any assumption on the "direction" of its parameters. Thus one can compute $y$ out of a given $x$, or, vice versa (as in Prolog)

This general use of user-defined constraints in JSetL is made possible thanks to:
- the use of **logical variables** as parameters
- the use of **unification** in place of equality and assignment
- the use of **nondeterminism** to compute multiple solutions for the same constraint
Another example - Implementing DCGs

Definite Clause Grammars (DCG) can be straightforwardly implemented in Java + JSetL using the nondeterminism support offered by JSetL

Ex.:

**EBNF:**

\[
\langle expr \rangle ::= \langle num \rangle | \langle num \rangle + \langle expr \rangle | \langle num \rangle - \langle expr \rangle
\]

**Prolog:**

expr(L, Remain) :- num(L, Remain).
expr(L, Remain) :- num(L, L1), L1 = [+|L2], expr(L2, Remain).
expr(L, Remain) :- num(L, L1), L1 = [-|L2], expr(L2, Remain).
num(L, Remain) :-L = [D|Remain], number(D).

**Java + JSetL:**

```java
public Constraint expr(LLList L, LLList Remain) {
    return new Constraint("expr", L, Remain);
}
```
Another example - Implementing DCGs – cont'd

```java
private void expr(Constraint c) {
    LList L = (LList)c.getArg(1);
    LList Remain = (LList)c.getArg(2);
    switch (c.getAlternative()) {
        // expr(L,Remain) :- num(L,Remain).
        case 0: {
            solver.addChoicePoint(c);
            solver.add(num(L,Remain));
            break;
        }
        // expr(L,Remain) :- num(L,L1), L1=[+|L2], expr(L2,Remain).
        case 1: {
            solver.addChoicePoint(c);
            LList L1 = new LList();
            LList L2 = new LList();
            solver.add(num(L, L1));
            solver.add(L1.eq(L2.ins('+')));
            solver.add(expr(L2, Remain));
            break;
        }
        ...
    }
}
```
Another example - Implementing DCGs – cont'd

If, for example, the expression to be parsed is

\[ 5 + 3 - 2 \]

which is represented by a logical list `tokenList` with value

\[
[ '5', '+', '3', '-', '2' ]
\]

and `sampleParser` is an instance of the class containing method `expr`, then the invocation

```
solver.check(sampleParser.expr(tokenList, LList.empty()))
```

will return `true`, while, if `tokenList` has value

\[
[ '5', '+', '3', '-' ]
\]

the same invocation to `sampleParser.expr` will return `false`.

CILC 2013
Conclusions & Future work

Shown that non-deterministic programming is conveniently exploitable also **within conventional O-O languages** such as Java.

This has been obtained by combining a number of different features offered by the Java library JSetL: set data abstractions, nondeterministic constraint solving, logical variables, unification, user-defined constraints.

In particular, general nondeterministic procedures can be defined in JSetL as new **user-defined constraints**.

In the near future:
Developing a **preprocessor** that allows the user to use more friendly syntax for expressing and dealing with nondeterminism in JSetL.

Adding **new facilities** for nondeterminism management, e.g., some form of “cut”, setof, ...
protected void user_code(Constraint c) throws NotDefConstraintException {
    if (c.getName().equals("expr"))
        expr(c);
    else if (c.getName().equals("num"))
        num(c);
    else if (c.getName().equals("number"))
        number(c);
    else {
        throw new NotDefConstraintException();
    }
}

public class ExprParser extends NewConstraintsClass {
    public ExprParser(SolverClass currentSolver) {
        super(currentSolver);
    }
    ...
}
Altri risolutori di vincoli

Disponibili risolutori efficienti per linguaggi con vincoli insiemistici forniti come librerie per linguaggi object-oriented:

Es.
    ILOG, Choco, JaCoP, Koalog, Gecode

Recentemente, comunità Java:
documento di specifica (JSR-331) per la definizione di
un'interfaccia (API) standard per la programmazione con vincoli, che include vincoli insiemistici

Proposte orientate principalmente alla soluzione efficiente di
problemi di soddisfacimento di vincoli (CSP), piuttosto che al supporto più in generale di programmazione (dichiarativa) con insiemi
Exploiting the embedded nondeterminism - examples

Ex.: find the maximum \( m \) of a set of integer numbers \( S \)

Declarative solution:

\[
\exists m \ (m \in S \land \forall y \in S, m \geq y)
\]

In Java + JSetL

```java
public static LVar max(Set S) throws Failure {
    LVar m = new LVar();
    LVar y = new LVar();

    solver.add(m.in(S)); // m \in S
    solver.forall(y,S,m.ge(y)); // forall(y \in S, m \geq y)

    solver.solve();
    return m;
}
```
(Multi-)Intervals and set variables

Aggiunta possibilità di trattare **intervalli** di interi, di insiemi e **multi-intervalli** di interi. Es.:

```
MultiInterval x = new MultiInterval(1,5);  // [1..5]
MultiInterval y = new MultiInterval(10);   // {10}
MultiInterval z = new MultiInterval(new Interval(20,30));
System.out.println(x.union(y).union(z));  // {1..5, 10, 20..30}
```

Integrazione in JSetL di un risolutore di vincoli su domini finiti di interi e su **domini finiti di insiemi di interi** (seguendo l'approccio di [Gervet '97] e [Azevedo 2007]). Es.:

```
MultiInterval lb = new MultiInterval();
MultiInterval ub = new MultiInterval(1, N);
SetLVar var = new SetLVar("S", lb, ub);  // S::[{}..[1..N]]
```
Esempi

Problema "colorazione mappa"

```java
public static void coloring(Set reg, Set map, Set col) throws Failure {
    Lvar x = new Lvar();
    Solver.add(reg.eq(col));    // reg = col
    Solver.forall(x, col, Set.empty.ins(x).nin(map));
        // For all x in col, x ∉ map
    Solver.solve();
    return;
}
```

Es.

regions: \{r1,r2,r3\} con r1, r2, r3 Lvar non inizializzate
map: Set contenente insiemi \{{r1,r2},{r2,r3}\}
colors: Set contenente stringhe (costanti) \{"blu","rosso"\}

regions: \{"blu","rosso","blu"\}
Esempi

Problema calcolo del “powerset” di un insieme

```java
public static Set powerset(Set s) throws Failure {
    Set r = new Set();
    Solver.add(r.subset(s));
    return Solver.setof(r);
}
```

```java
public static void main (String[] args) throws Failure {
    int[] s_elems = {1,2,3};
    Set s = new Set(s_elems); // s = {1,2,3}
    Set ps = new Set("powerset", powerset(s));
    ps.output();
    return;
}
```

output:

```
powerset = {{},{3},{2},{2,3},{1},{1,3},{1,2},{1,2,3}}
```
To obtain all possible values for \( n \), one could use the method `nextSolution` as follows:

```java
solver.check(ndTest.ndAssign(n));
do{
    System.out.println(n);
} while(solver.nextSolution());
```
In order to implement nondeterministic procedures as user-defined constraints we must also take into account the following:

- using (reference to) **logical objects as arguments** provides a simple way for the method to return results (the type of the method is **Constraint**)

- using logical objects as data within constraint methods is fundamental to allow backtracking to restore the correct state when necessary

- logical objects need to be manipulated only through constraints (e.g., \( \text{eq} \) constraints instead of assignments)