Nondeterministic Programming in Java with JSetL

Gianfranco Rossi and Federico Bergenti

Dipartimento di Matematica e Informatica, Università di Parma
{gianfranco.rossi | federico.bergenti}@unipr.it

Abstract. JSetL is a Java library that endows Java with a number of facilities that are intended to support declarative and constraint (logic) programming. In this paper we show how JSetL can be used to support general forms of nondeterministic programming in an object-oriented framework. This is obtained by combining different but related facilities such as logical variables, set data structures, unification, along with a constraint solver that allows the user to solve nondeterministic constraints, as well as to define new constraints using the nondeterminism handling facilities provided by the solver itself. Thus, the user can define her/his own general nondeterministic procedures as new constraints, letting the constraint solver handle them. The proposed solutions are illustrated by showing a number of concrete Java implementations using JSetL, including the implementation of simple Definite Clause Grammars.

Keywords. Nondeterministic programming, Constraint Programming, Set-based Programming, Java language.

1 Introduction

The problem of incorporating constructs to support nondeterminism into programming languages has been discussed at length in the past. Early references to this topic are [4] for a general overview, and [11] for an analysis of the problem in the context of functional programming languages. Logic programming languages, notably Prolog, strongly rely on nondeterminism. Their computational model is inherently nondeterministic (at each computation step, one of the clauses unifying a given goal is selected nondeterministically) and the programmer can exploit and control nondeterminism using the language features when defining her/his own procedures.

As regards imperative programming, however, only relatively few languages provide primitive constructs to support nondeterminism. An early example is SETL [10], a Pascal-like language endowed with sets, which provides, among others, a few built-in features to support backtracking (e.g., the `ok` and `fail` primitives). More recently, the programming language Alma-0 [1] [2] provides a comprehensive collection of primitive constructs to support nondeterministic

programming, such as the statements `orelse`, `some`, `forall`, `commit`, for creating choice points and handling backtracking. Also Python's `yield` mechanism—and, more generally, the coroutining mechanisms present in various programming languages—can be used as a way to explore the computation tree associated with a nondeterministic program.

Our goal in this paper is to explore the feasibility of a *library-based* approach to support nondeterministic programming in an object-oriented language. Specifically, our proposal is to exploit the nondeterministic constraint solver provided by JSetL [9], a Java library that combines the object-oriented programming paradigm of Java with valuable concepts of CLP languages, such as logical variables, partially specified list data structures, unification, constraint solving. Using this library the programmer can define nondeterministic procedures by exploiting either the nondeterminism embedded in the predefined constraints (in particular, in set constraints), or the possibility to define new user-defined nondeterministic constraints and handle them through the built-in constraint solver. We will illustrate our solution with a number of simple examples using Java with JSetL.

The paper is organized as follows. In Section 2 we show how JSetL can support nondeterminism through the use of built-in nondeterministic features, such as set constraints and the labeling mechanism. Section 3 briefly introduces general nondeterministic control structures and the relevant language constructs. In Section 4 we show how different nondeterministic control structures can be implemented in Java using the facilities for defining new constraints provided by JSetL. Finally, in Section 5 we show a more complete example of application of the facilities offered by JSetL to support nondeterministic control structures: the implementation of Definite Clause Grammars.

2 Embedded Nondeterminism

A convenient way to express nondeterminism in JSetL is by means of *set constraints*. As a matter of fact, nondeterminism is strongly related to the notion of set and set operations (see, e.g., [13] and [7]).

Sets can be defined in JSetL as instances of the class `LSet`. Elements of a `LSet` object can be of any type, including other `LSet` objects (i.e., *nested* sets are allowed). Moreover, sets denoted by `LSet` (also referred to as *logical sets*) can be *partially specified*, i.e., they can contain unknown elements, as well as an unknown part [3]. Single unknown elements are represented by unbound *logical variables* (i.e., uninitialized objects of the class `LVar`), whereas the unknown part of the set is represented by an unbound logical set (i.e., an uninitialized object of the class `LSet`). For example, the three statements:

```
LVar x = new LVar();
LSet r = new LSet();
LSet s = r.ins(x);
```

create an unbound logical variable `x`, an unbound logical set `r`, and a partially specified logical set `s` with an unknown element `x` and an unknown rest `r` (i.e., {`x` | `r`}, using a Prolog-like notation).

JSetL provides the basic operations on this kind of sets, such as equality (viz., *set unification* [6]), inequality, membership, cardinality, union, etc., in the form of primitive constraints, similarly to what provided by the Constraint Logic Programming language CLP($\mathcal{SET}$) [5]. A JSetL *constraint solver* is an instance of the class `SolverClass`. Basically, it provides methods for adding constraints to its *constraint store* (e.g., the method `add`) and to prove satisfiability of a given constraint (methods `check` and `solve`). If `solver` is a solver, Γ is the constraint stored in its constraint store (possibly empty), and `c` is a constraint, then `solver.check(c)` returns `false` if and only if $\Gamma \wedge$ `c` is unsatisfiable.

Solving equalities, as well as other basic set-theoretical operations, over partially specified sets may involve nondeterminism. For example, the equation $\{x, y\} = \{1, 2\}$, where x and y are unbound logical variables, admits two distinct solutions: $x = 1 \wedge y = 2$ and $x = 2 \wedge y = 1$. In JSetL, these solutions are computed nondeterministically by the constraint solver, using choice points and backtracking.

In the following example we exploit the nondeterminism embedded in set operations to provide a nondeterministic solution to the problem of printing all permutations of a set of integer numbers s. The problem can be modelled as the problem of unifying a (partially specified) set of $n = |s|$ logical variables $\{x_1, \ldots, x_n\}$ with the set s, i.e., $\{x_1, \ldots, x_n\} = s$. Each solution to this problem yields an assignment of (distinct) values to variables $x_1, \ldots, x_n$ that represents a possible permutation of the integers in s.

Example 1. (Permutations)

```
public static void allPermutations(LSet s) {
    int n = s.getSize();                // the cardinality of s
    LSet r = LSet.mkLSet(n);            // r = {x1,x2,...,xn}
    solver.check(r.eq(s));              // r = s
    do {
        r.printElems(' ');
        System.out.println();
    } while (solver.nextSolution());
}
```

The invocation `LSet.mkLSet(n)` creates a set consisting of `n` unbound logical variables. This set is unified, through the constraint `eq`, with the set of `n` integers `s`. This is done by invoking the method `check` of the current constraint solver `solver` (`solver` is assumed to be created outside the method `allPermutations`). The invocation `check(r.eq(s))` causes a viable assignment of values from `s` to variables in `r` to be computed. Values in `r` are then printed on the standard output by calling the method `printElems`.

Calling the method `nextSolution` allows checking whether the current constraint admits further solutions and possibly computing the next one. This

method exploits the backtracking mechanism embedded in the constraint solver: calling `nextSolution` forces the computation to go back until the nearest open choice point is encountered. Specifically, in the above example, solving `r.eq(s)` nondeterministically computes a solution to the set unification problem involving the two sets `r` and `s`. Thus, all possible rearrangements of the values in the given sets (i.e., all possible permutations) are computed and printed, one at a time.

The example illustrates also how the nondeterminism of the JSetL solver interacts with the usual features of the underlying imperative Java language.

Note that in this example nondeterminism is implemented simply by operations on sets and the nondeterministic search is completely embedded in the constraint solver. Since the semantics of set operations is usually well understood and quite intuitive, making nondeterministic programming the same as programming with sets can contribute to make the (non-trivial) notion of nondeterminism easier to understand and to use.

Whenever the problem at hand can be formulated as a Constraint Satisfaction Problem (CSP) over Finite Domains, solutions can be computed nondeterministically by exploiting the so-called *labeling* mechanism. Values to be assigned to variables of the CSP are picked up nondeterministically from the variable domains: if the selected assignment turns out to be not suitable, another alternative is then explored.

In JSetL this is obtained by using the constraint `label`. Solving the constraint `s.label()`, where `s` is a collection of logical variables, forces the program to nondeterministically generate an admissible assignment of values to variables in `s`, starting from the first variable in `s` and the first value in its domain (default labeling strategy in JSetL). This assignment is propagated to all the constraints in the constraint store and if none of them turns out to be unsatisfiable, then an assignment for the next variable in `s` is computed and propagated, and so on. As soon as a constraint in the store turns out to be unsatisfiable, backtracking occurs and a new assignment for the lastly assigned variable is computed. If a viable assignment for all the variables in `s` is finally found, then it represents a solution for the given CSP.

For example, the well-known *n-queens* problem, very often used as a sample problem for illustrating nondeterministic programming, can be easily modelled as a CSP and solved using constraints over Finite Domains and a final labeling phase to nondeterministically generate all possible solutions.

Unfortunately, not all problems whose solutions are naturally formulated as nondeterministic algorithms are also easily modelled as CSP. There are situations in which, in particular, the variable domains are difficult to bring under those supported by existing CP solvers, making the programming effort to model them in terms of the existing ones too cumbersome and sometimes quite *ad hoc*. On the other hand, the use of sets and set operations to model nondeterministic computations, as shown in this section, is not always feasible and/or convenient.

In conclusion, there are cases in which some more general programming abstractions to express and handle nondeterminism are required. We address this problem in the next sections.

3 Nondeterministic Control Structures

Dealing with general nondeterministic control structures requires primarily the ability to express and handle *choice points* and *backtracking*. This implies, first of all, that the notion of program computation is extended to allow distinguishing between computations that terminate with success and computations that terminate with failure. Basically, a computation *fails* whenever it executes, either implicitly or explicitly, a `fail` statement. Conversely, a finite, error-free computation *succeeds* if it does not fail. In response to a failure, the computation backtracks to the last open choice point. Choice points may be created by the programmer using suitable language constructs, such as the following `orelse` statement (borrowed from [1]):

$$\texttt{either } S_1 \texttt{ orelse } S_2 \ldots \texttt{orelse } S_n \texttt{ end}$$

which expresses a nondeterministic choice from among n statements $S_1 \ldots S_n$. More precisely, the computation of the `orelse` goes as follows: statement S_1 is executed first; if, at some point of the computation (possibly beyond the end of the `orelse` statement) a failure occurs, then backtracking takes place and the computation resumes with S_2 in the state it had when entering S_1; if a new failure occurs, then the computation backtracks and it resumes with S_3, and so on; if a failure occurs after executing S_n and no other open choice points do exist, then the computation definitively fails.

Let us briefly illustrate how to deal with general nondeterministic control structures with a simple example written using a C-like pseudo-language endowed with the `orelse` statement and a few other facilities to support nondeterministic programming. In the next section we will show how the same control structures can be implemented in Java with JSetL.

Given a list l of strings, split (all) the elements of l into two lists $l1$ and $l2$, such that the total length of the strings in $l1$ is equal to the total length of the strings in $l2$. For example, if l is `["I","you","she","we","you","they"]` a possible splitting of l is $l1$ = `["I","they","you"]` (total length = 8) and $l2$ = `["she","we","you"]` (total length = 8), whereas if `"she"` is replaced by `"he"` no splitting is feasible. Note that we are assuming that l can contain repeated elements and that strings can be picked up from l in any order. The problem can be solved by defining a function `split` that nondeterministically splits l into two lists $l1$ and $l2$, and then forcing `split` to generate (via backtracking) all possible pairs of lists $l1$ and $l2$ until a pair respecting the given condition is found. An implementation of this algorithm written in pseudo-code using the `orelse` construct is shown in Example 2.

Example 2. (List splitting—in pseudo-code)

```
split(l):
   either
      l is [];
      return ⟨[],[]⟩;
   orelse
      x is the first element and r the rest of l;
      ⟨r1, l2⟩ = split(r);
      return ⟨[x | r1], l2⟩;
   orelse
      x is the first element and r the rest of l;
      ⟨l1, r2⟩ = split(r);
      return ⟨l1, [x | r2]⟩;
   end;
```

where $[x \mid r1]$ (resp., $[x \mid r2]$) represents the list which is obtained by adding x as the first element to the list $r1$ (resp., $r2$). Therefore, if l is not the empty list, its first element is nondeterministically added to either the first sublist (second `orelse` alternative) or to the second sublist (third `orelse` alternative). If `sumLength(`l`)` is a function that computes the total length of all the strings in the list l, then the given problem is simply solved by calling `split(`l`)` and then requiring that the results of `sumLength(`$l1$`)` and `sumLength(`$l2$`)` are equal, that is:

```
⟨l1, l2⟩ = split(l);
sumLength(l1) == sumLength(l2);
```

Note that we are assuming that, in our pseudo-language, whenever an expression e is used as a statement, such as, for instance, `sumLength(`$l1$`) == sumLength(`$l2$`)` or l *is* `[]`, the following operational semantics is enforced: if e evaluates to `true` then continue; else fail.[1] Therefore, if the pair $\langle l1, l2\rangle$ computed by `split(`l`)` does not satisfy the condition `sumLength(`$l1$`) == sumLength(`$l2$`)`, then the computation backtracks to `split` and tries another open alternative created by the `orelse` statement, as long as at least one such alternative does exist.

This example shows also the typical interleaving between nondeterminism and recursion: each recursive call to `split` opens three branches in the nondeterministic computation of `split`. Executing the first `orelse` alternative, which represents the base of the recursion, corresponds to reaching a leaf in the computation tree, i.e. a possible solution.

The domain of discourse, in this example, is that of lists of strings. Moreover we do not make any assumption on the length of the lists, on the presence of duplicated elements in them, and on the length of the strings composing them. Trying to encode this domain in terms of the usual constraint domains and trying to restate the problem as a CSP, e.g. over the domain of integer numbers, though feasible in principle, may lead to rather involved programs in practice. On the other hand, trying to restate that problem as a set-theoretical one, in order to

[1] This goes much like the "boolean expressions as statement" feature of Alma-0 [1].

exploit the nondeterminism involved in set constraints as shown in Section 2, may be hindered, at least, by the presence of duplicates in the input sequence.

As mentioned in Section 1, very few programming languages support the above mentioned nondeterministic constructs as *primitive* features. Extending the language with *primitive* constructs that offer such support is, indeed, quite demanding in general. Our goal in this paper is to explore the feasibility of a *library-based* approach, where features to support nondeterministic programming are implemented on the top of an high-level language, namely Java, by exploiting the language abstraction mechanisms, without requiring any modification to the language itself.

We will illustrate this solution in the next sections with a number of simple examples, using the Java library JSetL.

4 Implementing Nondeterministic Control Structures in JSetL

As shown in Section 2, JSetL embeds nondeterminism at various levels. In particular, set constraints, as well as the labeling mechanism, are inherently nondeterministic. Availability of built-in nondeterministic constraints, however, is not sufficient to ensure the general kind of nondeterminism we would like to have.

The solution that we propose in order to circumvent such difficulties is based on the availability in JSetL of a nondeterministic constraint solver and the possibility for the programmer to define her/his own (nondeterministic) constraints. Those methods that require the use of nondeterminism are defined as new user-defined constraints. Within these methods the programmer can exploit facilities offered by JSetL for creating and handling choice-points. When solving these constraints the solver will explore the different alternatives using backtracking.

User-defined constraints in JSetL are defined as part of a user class that extends the abstract class `NewConstraintsClass`. The actual implementation of user-defined constraints requires some programming conventions to be respected, as shown in the following example.

Example 3. (Implementing new constraints) Define a class `MyOps` which offers two new constraints `c1(o1,o2)` and `c2(o3)`, where `o1`, `o2`, `o3` are objects of type `t1`, `t2`, and `t3`, respectively.

```
public class MyOps extends NewConstraintsClass {
   public MyOps(SolverClass currentSolver) {
       super(currentSolver);
   }
   public Constraint c1(t1 o1, t2 o2) {
           return new Constraint("c1", o1, o2);
   }
   public Constraint c2(t3 o3) {
           return new Constraint("c2", o3);
```

```
    }
    protected void user_code(Constraint c)
    throws Failure, NotDefConstraintException {
        if (c.getName().equals("c1")) c1(c);
        else if(c.getName().equals("c2")) c2(c);
        else throw new NotDefConstraintException();
    }
    private void c1(Constraint c) {
        t1 x = (t1)c.getArg(1);
        t2 y = (t2)c.getArg(2);
        //implementation of constraint c1 over objects x and y
    }
    private void c2(Constraint c) {
        t3 x = (t3)c.getArg(1);
        //implementation of constraint c2 over object x
    }
}
```

The one-argument constructor of the class `MyOps` initializes the field `solver` of the super class `NewConstraintsClass` with (a reference to) the solver currently in use by the user program.

The other public methods simply construct and return new objects of class `Constraint`. This class implements the *atomic constraint* data type. All built-in constraint methods implemented by JSetL (e.g., `eq`, `neq`, `in`, etc.) return an object of class `Constraint`. Each different constraint is identified by a string name (e.g., `"c1"`), which can be specified as a parameter of the constraint constructor.

The method `user_code`, which is defined as abstract in `NewConstraintsClass`, implements a "dispatcher" that associates each constraint name with the corresponding user-defined constraint method. It will be called by the solver during constraint solving.

Finally, the private methods, such as `c1` and `c2`, provide the implementation of the new constraints. These methods must, first of all, retrieve the constraint arguments, whose number and type depend on the constraint itself. We will show possible implementations of such methods (using nondeterminism) in Examples 4 and 6.

Once objects of the class containing user-defined constraints have been created, one can use these constraints in the same way as the built-in ones: user-defined constraints can be added to the constraint store using the method `add` and solved using the `SolverClass` methods for constraint solving. For example, executing the statements

```
MyOps myOps = new MyOps(solver);
solver.solve(myOps.c1(o1,o2));
```

first creates an object of type `MyOps`, called `myOps`, then it creates the constraint `"c1"` over two objects `o1` and `o2` by calling `myOps.c1(o1,o2)`, finally it adds this constraint to the constraint store and solves it by calling the method `solve`

of `solver`. Solving the constraint `"c1"` will cause the solver to call the concrete implementation of the method `user_code` provided by `myOps`, and consequently to execute its method `c1`.[2]

User-defined constraints in JSetL can implement nondeterministic procedures by exploiting special features offered by the JSetL constraint solver. Defining nondeterministic constraints in JSetL, however, requires some additional considerations to be taken into account.

First of all note that methods defining user-defined constraints must necessarily return an object of type `Constraint`. Thus, any other result possibly computed by the method must be returned through parameters. The use of unbound *logical objects*, i.e., logical variables as well as logical sets and lists, as arguments of the user-defined constraints provides a simple solution to this problem.

More generally, the use of logical objects is fundamental in JSetL when dealing with nondeterminism. As a matter of fact, if an object is involved in a nondeterministic computation then it is necessary to restore the status it had before the last choice point whenever the computation backtracks and tries a different alternative. In JSetL this is obtained by allowing the solver to automatically save and restore the global status of all *logical objects* involved in the computation. Since a logical object is characterized by the fact that its value, if any, can not be changed through side-effects, saving and restoring the status of logical objects is a relatively simple task for the solver. Hence, we will always use logical objects, in particular logical variables, for all those objects that are involved in nondeterministic computations.

As a consequence of this choice we can not manipulate logical objects by using the usual imperative statements (e.g., the assignment), but we will always need to use constraints to deal with them. In particular, the equality constraint l`.eq(`v`)` is used to *unify* a logical variable l with a value v. If l is unbound, this simply amounts to binding l to v. Once assigned, however, the value v is no longer modifiable.

As an example let us consider the implementation of the nondeterministic function `split(`l`)` shown in Example 2. This function can be implemented in JSetL by a user-defined constraint `split(l,l1,l2)`. Note that, here we are replacing a function with the corresponding *relation*: in fact, `split(l,l1,l2)` defines a ternary relation whose elements are those triples $\langle$`l,l1,l2`$\rangle$, with `l`, `l1` and `l2` belonging to the domain of lists, such that all elements of `l` are split into two lists `l1` and `l2`.

As noted above, variables dealt with by nondeterministic constraints are conveniently represented in JSetL as logical objects. Thus, we represent the lists `l`, `l1`, and `l2` of the constraint `split` as JSetL's logical lists (i.e., objects of the class `LList`) and we manipulate them through constraints over lists.

[2] Note that the constructor of the super class `NewConstraintsClass`, which is invoked when `myOps` is created, provides for storing (a reference to) itself within the specified solver, so making the latter able to invoke the method `user_code` of the created object `myOps`.

Example 4. (`split` in JSetL) Define a constraint `split(l,l1,l2)` which is true if all elements of the list `l` are split into two lists `l1` and `l2`.

```
public Constraint split(LList l,LList l1,LList l2) {
   return new Constraint("split",l,l1,l2);
}
private void split(Constraint c) throws Failure {
   LList l = (LList)c.getArg(1);
   LList l1 = (LList)c.getArg(2);
   LList l2 = (LList)c.getArg(3);
   switch(c.getAlternative()) {
   case 0:
      solver.addChoicePoint(c);
      solver.add(l.eq(LList.empty()));     // l is []
      solver.add(l1.eq(LList.empty()));    // l1 is []
      solver.add(l2.eq(LList.empty()));    // l2 is []
      break;
   case 1: {
      solver.addChoicePoint(c);
      LVar  x = new LVar();
      LList r = new LList();
      LList r1 = new LList();
      solver.add(l.eq(r.ins(x)));   // 1st element (x) and rest (r) of l
      solver.add(split(r,r1,l2));   // split r into two lists r1 and l2
      solver.add(l1.eq(r1.ins(x))); // l1 is [x|r1]
      break; }
   case 2: {
      LVar  x = new LVar();
      LList r = new LList();
      LList r2 = new LList();
      solver.add(l.eq(r.ins(x)));   // 1st element (x) and rest (r) of l
      solver.add(split(r,l1,r2));   // split r into two lists l1 and r2
      solver.add(l2.eq(r2.ins(x))); // l2 is [x|r2]
     break; }
    }
}
```

`split` implements the nondeterministic construct `orelse` by using the methods `getAlternative` and `addChoicePoint`. The invocation `c.getAlternative()` returns an integer associated with the constraint `c` that can be used to count the nondeterministic alternatives within this constraint. Its initial value is 0. Each time the constraint `c` is re-considered due to backtracking, the value returned by `c.getAlternative()` is automatically incremented by 1. The invocation `solver.addChoicePoint(c)` adds a choice point to the alternative stack of the current solver. This allows the solver to backtrack and re-consider the constraint `c` if a failure occurs subsequently.

Note that the JSetL implementation of the method `split` closely resembles the abstract definition in pseudo-code of the function `split` given in Section 3. In particular, lists are implemented as `LList` objects, and the addition and extraction of elements from such lists is performed through the JSetL constraint `eq`.

Specifically, `l.eq(r.ins(x))` is true if `l` is the list composed by an element `x` and a remaining part `r`. If `l` is known and `x` and `r` are not, solving `l.eq(r.ins(x))` amounts to computing the first element `x` and the rest `r` of the list `l`, whereas if `x` and `r` are known and `l` is not, `l.eq(r.ins(x))` can be used to build the list `l` out of its parts `x` and `r`.

Finally, note that the recursive call to `split(`r`)` in the abstract definition of the function `split` is replaced by the (recursive) posting of the constraint `split(r,r1,l2)` (or `split(r,l1,r2)`) in the above concrete implementation.

As a sample use of `split`, if `l` is the `LList` with value `["I","you","she","we","you","they"]`, `sumLength(l,n)` is a user-defined (deterministic) constraint that implements the function `sumLength(`l`)` of Example 2, `listOps` is an instance of the class that extends `NewConstraintsClass` containing `split` and `sumLength`, and `l1`, `l2` are unbound logical lists and `n`, `m` are unbound logical variables, then executing the following fragment of code

```
solver.add(listOps.split(l,l1,l2));
solver.add(listOps.sumLength(l1,n));
solver.add(listOps.sumLength(l2,m));
solver.check(m.eq(n));
```

will bind `l1` to `["you","I","they"]`, `l2` to `["she","you","we"]`, and `n` and `m` to 8.

Remark 1. The use of relations in place of functions, along with the use in their implementation of a number of specific features provided by JSetL, have another important consequence on the usability of user-defined constraint methods.

Let us consider a function $y = f(x)$ and a possible call to it, $z = f(a)$. In JSetL one can define a constraint $c_f(x, y)$ which represents the relation $R_f = \{\langle x, y\rangle : y = f(x)\}$ and then solve the constraint $c_f(a, z)$. Solving this constraint actually amounts to checking whether $\langle a, z\rangle \in R_f$, for some z. While calling $f(x)$ to compute y implies assuming x to be the input parameter and y the output, solving $c_f(x, y)$ does not make any assumption on the "direction" of their parameters. Thus, one can compute y out of a given x, or, vice versa, x out of a given y, or one can test whether the relation among two given values x and y holds, or one can compute any of the pairs $\langle x, y\rangle$ belonging to R_f. Hence, the same method can be used to implement different, though related, functions.[3]

This general use of user-defined constraints in JSetL is made possible thanks to the availability of a number of different facilities to be used in the constraint implementation. Specifically,

- 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.

[3] It worth emphasizing here the similarity with Prolog or, more to the point, with Prolog-based CP languages.

Note that the fact that a logical variable acts as an input or as an output parameter depends on the fact it is bound or not when the method is called. In particular, unbound variables can be easily used to obtain output parameters.

Moreover, if the value bound to a variable is a partially specified aggregate, e.g. a logical list, then it can act simultaneously as input and as output, i.e. as an input-output parameter. For example, let us consider the fragment of code shown at the end of Example 4. If we want to say, for instance, that `l2` must contain two repeated elements, then the above statements can be preceded by the following declarations

```
LVar x = new LVar();
LList l2 = new LList().ins(x).ins(x);
```

In this way `split` is called with its third argument bound to the partially specified list `[x,x|_]` instead of being left unbound. Thus, solving `split(l,l1,l2)` will bind `l1` to `["I","she","they"]` and `l2` to `["you","you","we"]`.

5 Implementing DCGs

In this section we show a more complete example of application of the facilities offered by JSetL to support nondeterminism: the implementation of Definite Clause Grammars [8].

A Definite Clause Grammar (DCG) is a way to represent a context-free grammar as a set of first-order logic formulae in the form of definite clauses. As such, DCGs are closely related to Logic Programming, and tools for dealing with DCGs are usually provided by current Prolog systems. Given the DCG representation of a grammar one can immediately obtain a parser for the language it describes by viewing the DCG as a set of Prolog clauses and using the Prolog interpreter to execute them.

In this section we show how DCGs can be conveniently used also in the context of more conventional languages, such as Java, provided the language is equipped with a few features that are fundamental to support DCGs processing, namely (logical) lists and nondeterminism. We prove this claim by showing how DCGs can be encoded and processed using Java with JSetL.

Consider the following excerpt of a grammar of constant arithmetic expressions

$$\langle expr \rangle ::= \langle num \rangle | \langle num \rangle + \langle expr \rangle | \langle num \rangle - \langle expr \rangle$$

Assume that input to be parsed is represented as a list of numerals and symbols. For example, `[8, +, 2, -, 7]` is a valid $\langle expr \rangle$.

This grammar may be encoded in terms of first-order logic formulae in clausal form in the following way: create one predicate for each non-terminal in the grammar and define each predicate using one clause for each alternative form of the corresponding non-terminal. Each predicate takes two arguments, the first being the list representation of the input stream, and the second being instantiated to the list of input elements that remain after a complete syntactic

structure has been found. As an example, the above grammar can be written as a DCG as follows (using a pure Prolog notation).

Example 5. (A DCG for $\langle expr \rangle$)

```
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).
```

where the predicate `number(D)` is true if `D` is a numeric constant.[4]

This grammar representation constitutes an executable Prolog program that can be immediately used as a top-down parser for the denoted language. Using this program we can prove that, for example,
`expr([1, +, 2, -, 3], [])`
is true (i.e., `1+2-3` is a valid arithmetic expression), while
`expr([1, +, 2, -], [])`
is false.

The DCG shown in Example 5, that we have written as a Prolog program, can be implemented with a relatively little effort as a JSetL program as well. Each predicate corresponding to a non-terminal in the grammar is implemented as a new JSetL constraint, that is a method of a class extending the class `NewConstraintsClass`. These methods exploit the nondeterministic features provided by JSetL to support the nondeterministic choice from among different clauses for the same predicate. List data structures are implemented using JSetL logical lists, that is objects of the class `LList`. In particular, partially specified lists with an unknown rest (i.e., $[o \,|\, l]$, l unbound) can be constructed using the method `ins` and accessed through unification. The complete JSetL implementation of the DCG shown above is given in Example 6.

Example 6. (Implementing the DCG for $\langle expr \rangle$ in JSetL)

```
public class ExprParser extends NewConstraintsClass {
      public ExprParser(SolverClass currentSolver) {
         super(currentSolver);
      }
      public Constraint expr(LList L, LList Remain) {
         return new Constraint("expr", L, Remain);
      }
```

[4] Special syntax exists in current Prolog systems that allows EBNF-like specification of DCGs. For instance, the second clause of Example 5 can be written in Prolog alternatively as `expr --> num, [+], expr`. The Prolog interpreter automatically translates this special form to the pure clausal form used in Example 5.

```
public Constraint num(LList L, LList Remain) {
   return new Constraint("num", L, Remain);
}
public Constraint number(LVar n) {
   return new Constraint("number", n);
}

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();
  }
}

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;
    }
    // expr(L, Remain) :- num(L, L1), L1 = [-|L2], expr(L2, Remain).
    case 2: {
      LList L1 = new LList();
      LList L2 = new LList();
      solver.add(num(L, L1));
      solver.add(L1.eq(L2.ins('-')));
      solver.add(expr(L2, Remain));
    }
  }
}
```

```
      private void num(Constraint c) {
        LList L = (LList)c.getArg(1);
        LList Remain = (LList)c.getArg(2);
        LVar D = new LVar();
        solver.add(L.eq(Remain.ins(D)));
        solver.add(number(D));
      }

      private void number(Constraint c) {
        LVar n = (LVar)c.getArg(1);
        if (n.getValue() instanceof Integer)
              return;
        else
              c.fail();
      }
}
```

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 `ExprParser`, 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`.

Actions to be performed when a non-terminal has been successfully reduced (e.g., in order to evaluate the parsed expression or to generate the corresponding target code) can be easily added to a DCG by adding new arguments to predicates defining non-terminals and new atoms at the end of the body of the corresponding clauses. Accordingly, the JSetL implementation of a DCG can be easily extended by adding new arguments and suitable statements to the user-defined constraints implementing the non-terminals.

6 Conclusions and future work

In this paper we have made evident, through a number of simple examples, that nondeterministic programming is conveniently exploitable also within conventional O-O languages such as Java. We have obtained this 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, taking advantage of the facilities for expressing and handling nondeterminism provided by the solver.

The JSetL library, along with the source code of all the sample programs shown in this paper, can be downloaded from the JSetL's home page at `http://cmt.math.unipr.it/jsetl.html`.

As a future work we plan to identify the minimal extensions to be made to the JSetL's solver to make it capable of supporting, with the same approach outlined in this paper, other nondeterministic control structures e.g. the ones described in [1] and [12].

Acknowledgments

This work has been partially supported by the G.N.C.S. project "Specifica e verifica di algoritmi tramite strumenti basati sulla teoria degli insiemi". Special thanks to Luca Chiarabini who took part and stimulated the preliminary discussions on this work, and to Andrea Longo who contributed to the development of the JSetL based implementation of DCGs.

References

1. K.R. Apt, J. Brunekreef, V. Partington, A. Schaerf (1998) Alma-0: An Imperative Language that Supports Declarative Programming. ACM Transactions on Programming Languages and Systems (TOPLAS), Vol. 20, No. 5, 1014–1066.
2. K.R. Apt, A. Schaerf (1999) The Alma Project, or How First-Order Logic Can Help Us in Imperative Programming. In E.-R. Olderog, B. Steffen, Eds., *Correct System Design*, *LNCS*, Springer-Verlag, v. 1710, 89–113.
3. F. Bergenti, L. Chiarabini, and G. Rossi (2011) Programming with Partially Specified Aggregates in Java. *Computer Languages, Systems & Structures*, Elsevier, 37(4), 178–192.
4. J. Cohen (1979) Non-Deterministic Algorithms. Computing Surveys, Vol. 11, No. 2, 79–94.
5. A. Dovier, C. Piazza, E. Pontelli, and G. Rossi (2000) Sets and Constraint Logic Programming. *ACM Transactions on Programming Languages and Systems*, 22(5):861–931.
6. A. Dovier, E. Pontelli, and G. Rossi (2006) Set unification. *Theory and Practice of Logic Programming*, 6:645–701.
7. S.H. Mirian-HosseinaAbadi, M.R. Mousavi (2002) Making Nondeterminism Explicit in Z. Proceedings of the Iranian Computer Society Annual Conference (CSICC 02), Tehran, Iran.
8. F.C.N. Pereira, D.H.D. Warren (1980) Definite clause grammars for language analysis - A survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence* 13, 3, 231-278.
9. G. Rossi, E. Panegai, and E. Poleo (2007) JSetL: A Java Library for Supporting Declarative Programming in Java. *Software-Practice & Experience*, 37:115-149.
10. J. T. Schwartz, R. B. K. Dewar, E. Dubinsky, and E. Schonberg (1986) *Programming with sets, an introduction to SETL*. Springer-Verlag.
11. H. Sondergaard, P. Sestoft (1992) Non-Determinism in Functional Languages. The Computer Journal, 35(5), 514-523.
12. P. Van Hentenryck, L. Michel (2006) Nondeterministic Control for Hybrid Search. Constraints, Vol. 11, No. 4, 353– 373.
13. M. Walicki, S. Meldal (1993) Sets and Nondeterminism. Presented at ICLP'93 Post-Conference Workshop on Logic Programming with Sets (`http://people.math.unipr.it/gianfranco.rossi/sets/body_workshop.html`), Budapest.