JSetL: A Java Library for Supporting Declarative Programming in Java*

G. Rossi†, E. Panegai, E. Poleo
Dipartimento di Matematica
Università di Parma, Parma (Italy)

Abstract

In this paper we present a Java library—called JSetL—that offers a number of facilities to support declarative programming like those usually found in logic or functional declarative languages: logical variables, list and set data structures (possibly partially specified), unification and constraint solving over sets, nondeterminism. The paper describes the main features of JSetL and it shows, through a number of simple examples, how these features can be exploited to support a real declarative programming style in Java.

Keywords: Declarative Programming; Constraint Programming; Set Constraints; Java; Nondeterminism.

1 Introduction

Declarative programming (DP) is a programming paradigm in which, generally speaking, the programmer provides the *what* the program does, rather than the *how* it does. In DP the solution is described in terms of a set of conditions (relationships between variables), rather than as a list of instructions to execute in a specified order as we are accustomed to in imperative programming. The interpretation of the solution description is left up to an executor that performs the specific steps needed to produce a result according to some fixed strategy.

Declarative programming has been exploited in a number of different application areas including, but not limited to, artificial intelligence, natural language processing, databases, configuration management, constraint-satisfaction problems, Web services, hardware design. The main advantages usually awarded to the DP paradigm are easiness of program development and understanding, conciseness, reusability, and implicit parallelism.

Declarative programming languages are either general or special purpose. Among special purpose languages which are generally considered declarative we can mention: SQL, the Interface Description Language (IDL), the Web Services Description

*A preliminary abridged version of this paper, titled "JSetL: Declarative Programming in Java with Sets", appeared in the proceedings of the 2004 ACM International Conference on Computing Frontiers, ACM Press, 2004.

†**Contact Author**: Gianfranco Rossi, Dipartimento di Matematica, Università di Parma, Parco Area delle Scienze, 53/A, 43100 Parma (Italy), +39 (0521) 90.6909 (Phone), +39 (0521) 90.6950 (Fax), `gianfranco.rossi@unipr.it`.

Language (WSDL)—as well as many of the dialects being created based on the XML format—, MetaPost, various hardware design languages, spreadsheets-based languages, and so on. *General purpose DP languages* mainly include logic programming languages (such as Prolog and Mercury) and functional programming languages (such as Scheme and Haskell), as well as functional + logic and multi-paradigm languages (such as Curry, Escher and Oz).

More recently, a number of proposals have been put forward which aim at supporting general purpose declarative programming in the context of traditional, possibly object-oriented (OO), programming languages. As a matter of fact, while it is undeniable that DP fits into logic and functional programming very naturally, it is also a reality that most real-world software development is still done using traditional, possibly object-oriented, programming languages, such as C++ and Java. Integrating DP into a more conventional framework has the potential advantage to allow the programmer to exploit the DP paradigm while retaining all the advantages of constructs for programming and software structuring that are typical of conventional programming languages.

Some of these proposals define *new* programming languages (or extensions of existing languages) that provide, among others, DP features as "first-class citizens" of the language itself. Among them we can mention Alma-0 [2], Singleton [21], DJ (Declarative Java) [24, 25], and Flow Java [13]. However, the design and development of a new language—or even the extension of an existing one—is surely a difficult task and the resulting system is likely to be difficult to integrate with other existing systems and to be accepted by programmers.

An alternative to the new language approach is the *library-based approach* where the new facilities are added to a programming language (the host language) as part of a library, written in the language itself. This approach is largely adopted by those proposals aiming at augmenting an existing language with constraints. *Constraint programming* is a form of declarative programming. As a matter of fact, constraints provide a powerful tool for stating solutions as sets of conditions (e.g., equalities and inequalities) over the selected domains, which are then solved by using domain specific knowledge, with no concern about the order in which constraints occur in the program.

Initially the host languages used to embed constraint languages were in most cases logic-based languages (such as in CHIP[9] and in ECLiPSe [14]). More recently, libraries for supporting constraint handling have been developed also for conventional OO programming languages. One of the best known proposals following this approach is the ILOG Solver [20, 16]. In this system, constraints and logical variables are handled as objects and are defined within a C++ class library. Thanks to the encapsulation and operator overloading mechanisms, programmers can view constraints almost as if they really were part of the language. Among other proposals that take a similar approach we can mention JSolver [5], Choco [4], Koalog [18], JACK [1], and the Java Constraints Library (JCL) [17], all based on the Java programming language.

The above-mentioned solutions concentrate their attention mainly on constraint expressiveness and efficient constraint solving. While it is undeniable that constraint handling constitutes a valuable support to declarative programming, it also true that constraints are not the only way to support declarative programming in a broader sense. At our knowledge, no proposal has investigated the problem of enriching an

OO language with a comprehensive collection of library facilities to support general purpose declarative programming.

In this paper we present a Java library—called JSetL—that endows Java with a number of facilities to support general purpose declarative programming like those usually found in Constraint Logic Programming (CLP) languages: logical variables, unification, recursive data structures, constraint solving, nondeterminism. Specifically, JSetL provides most of the facilities provided by the *CLP language with sets* CLP($\mathcal{SET}$) [10]. In particular JSetL allows the programmer to create sets (possibly partially specified) and to manipulate them through constraints. Sets also provide a natural way to express nondeterminism and to collect all solutions for problems involving large solution spaces.

All these facilities are included in the JSetL library in a uniform way by exploiting the Java abstraction mechanisms. We show, with a number of simple examples, how the JSetL facilities can be combined with the usual Java constructs to obtain Java programs that mix declarative code with conventional imperative code.

Differently from other library-based proposals cited above we do not restrict our attention to constraints: constraints in JSetL are just one of the facility that our library provides to support declarative programming. Our attention is mainly devoted to find out a "good" collection of facilities to support general purpose declarative programming. Conversely, computation efficiency is not a primary requirement of our work. This is in some sense the counterpart of the flexibility and generality of our proposal.

The paper is organized as follows. In Section 2 we give an informal presentation of JSetL and we show a simple Java program using JSetL. In Section 3 we introduce the fundamental data structures of JSetL, namely logical variables, sets and lists. In Section 4, we describe the (set) constraint handling facilities supported by our library and we show how constraint solving can be accomplished, and how it interacts with the usual notion of program computation. The fundamental notion of nondeterminism and its relationship with sets is addressed in Section 5. In Section 6 we give a few definitions which allow to associate a declarative reading to Java methods using JSetL. In Section 7 we show how user defined constraints can be introduced in a program and how they can be exploited. Finally, in Section 8 we compare our proposal with some related work, and in Section 9 we draw some conclusion and discuss future work.

2 A short overview of JSetL

In this section, we briefly recall the most notable features of JSetL, that will be described in detail in the next sections. Moreover we show a simple example to give the flavor of the programming style supported by our library. A more detailed and complete presentation of all the features provided by JSetL can be found in [19].

2.1 Main features

- *Logical variables.* JSetL supports the notion of logical variable, like that usually found in logic and functional programming languages. Logical variables can be either initialized or *uninitialized*. The value of a logical variable in JSetL can be of any type. A value can be assigned to a (uninitialized) logical variable as the result of processing some constraint involving the variable, in particular, equality constraints. Constraints can be used to set as well as to inspect the value of a logical variable, but not to modify it.

- *Lists* and *sets*. JSetL provides list and set data structures, like those found, for instance, in [12]. List and set objects are dealt with as logical variables, but their values are restricted to list and set objects. Moreover, methods are provided to build lists/sets out of their elements. The main difference between lists and sets is that, while in lists the order and repetitions of elements are important, in sets order and repetitions of elements do not matter. Both lists and sets can be *partially specified*, i.e., they can contain uninitialized logical variables in place of single elements or as part of the list/set.

- *Unification*. JSetL provides unification between logical variables—as well as between list and set objects—basically in the form of equality constraints: `O1.eq(O2)`, where `O1` and `O2` are either logical variables, or list, or set objects, unifies `O1` and `O2`. Unification between two objects can be used either to test equality between the objects or to assign values to uninitialized logical variables (lists/sets) possibly occurring in the objects. Solving equality constraints involving sets implies in general the ability to deal with *set unification* (cf., e.g., [12]).

- *Constraints*. Basic set-theoretical operations, as well as equalities, inequalities and integer comparison expressions, are dealt with as *constraints*. Constraint expressions are evaluated using a powerful (set) constraint solver, which is basically the same developed for CLP($\mathcal{SET}$) [10]. Constraints are first added to the *constraint store*, then one can check their satisfiability by calling the solver. Constraint checking can be performed even if the constraints contain uninitialized (logical) variables, which allows us to compute with partially specified data. The order in which constraints are added to the constraint store is completely immaterial.

- *Nondeterminism*. Constraint solving in JSetL is inherently nondeterministic: the order in which constraints are solved does not matter (*don't care nondeterminism*); solutions for set constraints (e.g., equality, membership,...) are computed nondeterministically, using choice points and backtracking (*don't know nondeterminism*). JSetL provides also a simple way—using the method `setof`—to explore the search space of a nondeterministic computation and to collect into a set all the computed solutions for a specified logical variable `x`.

- *User-defined constraints*. JSetL allows the user to define new constraints. All definitions must be provided as part of the class `NewConstraints` within the JSetL package. A number of facilities for nondeterminism handling are also provided which can be exploited in the implementation of the new constraints. User-defined constraints can be dealt with as the built-in constraints: they can be added to the constraint store and solved by invoking the constraint solver.

All these facilities strongly contribute to support a real declarative programming style. In particular, the use of partially specified dynamic data structures, the ability to deal with constraint expressions disregarding the order in which they are encountered and the instantiation of the logical variables possibly occurring in them, as well as the nondeterminism naturally supported by operations over sets, are fundamental features to allow the language to be used as a highly declarative modeling tool.

2.2 Programming with JSetL: an introductory example

In order to illustrate the programming style supported by our library and to give an intuitive idea of how its distinguishing features can be used we show a simple example of a Java program using JSetL.

Problem: Compute and print the maximum of a set of integers `s`.

A truly declarative solution for this problem can be stated as follows: an element `x` of `s` is the maximum of `s` if for each element `y` of `s` it holds that `y` $\leq$ `x`. The program below shows how this solution can be immediately implemented in Java using JSetL.

Observe that here we are deliberately assuming that execution efficiency is not a primary requirement. Indeed, JSetL is mainly conceived as a tool for rapid software prototyping, where easiness of program development and program understanding prevail over efficiency.

```
class Max {
    static SolverClass Solver = new SolverClass();
    public static Lvar max(Set s) throws Failure {
        Lvar x = new Lvar();
        Lvar y = new Lvar();
        Solver.add(x.in(s));
        Solver.forall(y,s,y.leq(x));
        Solver.solve();
        return x;
        }

    public static void main (String[] args)
    throws IOException, Failure {
        int[] sample_set_elems = {1,6,4,8,10,5};
        Set sample_set = new Set(sample_set_elems);
        System.out.print(" Max = ");
        max(sample_set).print();
        }
    }
```

For the sake of simplicity we assume that the set of integers is directly supplied by the program (instead of being read for instance from a file). Hence we will focus on the definition of the method `max` that computes the maximum of `s`.

The first statement in the class `Max` creates the solver (simply named `Solver`) as an instance of the class `SolverClass`. `x` and `y` in the method `max` are two logical variables and both are uninitialized. Invocation of the method `add` of `Solver` adds the constraint `x.in(s)` (i.e., `x` $\in$ `s`) to the current constraint store. This constraint is evaluated to `true` if `s` is a set and `x` belongs to `s`. If `x` is uninitialized when the expression is evaluated this amounts to *nondeterministically* assign an element of `s` to `x`. Invocation of the method `forall` allows us to add to the constraint store a new constraint `y.leq(x)` (i.e., `y` $\leq$ `x`) for each `y` belonging to `s`. As soon as the method `solve` is invoked the constraint solver checks whether the current collection of constraints in the constraint store is satisfiable or not. If it is, the invocation of the method `solve` terminates with success. In this case, the value of `x` represents the integer we are looking for and it is returned as the result of `max`. If, on the contrary, one of the constraints in the constraint store is evaluated to false, backtracking takes place and the computation goes back till the nearest choice point. In this case, the nearest and only choice point is the one created by the constraint `x.in(s)`. Its execution will bind nondeterministically `x` to each element of `s`, one after the other. If all values of `s` have been attempted, there is no further alternative to explore and the computation of `max` terminates raising the exception `Failure`. If no **catch** clause for this exception is provided, the whole computation terminates reporting a failure.

Executing the program with the sample set of integers declared in the method **main** causes the message `Max = 10` to be printed to the standard output.

3 Logical variables and composite data objects

JSetL provides logical variables and two new kinds of data structures: sets and lists. These new features are implemented by three new classes, `Lvar`, `Lst`, and `Set`, for creation and manipulation of logical variables, lists and sets, respectively.

Lists and sets represent two different data abstractions: while in lists the order and repetitions of elements are important, in sets order and repetitions of elements do not matter. Thus, for instance, {1,2}, {2,1}, and {2,1,2} all denote the same set, while the analogous list expressions denote different lists. Although sets and lists can be often used interchangeably, there are cases—especially when sets and lists contain unknown elements—in which one choice can be more appropriate than the other. For example, if one wants to represent an undirected graph, arcs are likely to be represented as sets of two elements rather than as lists. As another example, if one wants to state that a collection A must contain the number 1, disregarding the position of 1 in A, A is conveniently represented as a partially specified set containing 1 (cf., e.g., set `s2` in Example 3.11).

3.1 Logical variables

Definition 3.1 *(Logical variable) A* logical variable *is an instance of the class* ***Lvar****, created by the declaration*

```
Lvar VarName = new Lvar(VarNameExt,VarValue);
```

where VarName *is the variable name,* VarNameExt *is an optional external name, and* VarValue *is an optional value for the variable.*

The *external name* is a string value which can be useful when printing the variable and the possible constraints involving it (if omitted, a default name of the form `"Lvar_`n`"`, where n is a unique integer, is assigned to the variable automatically). `Lvar` are not typed. The value associated with a logical variable—called an ***Lvar-value***—can be either a primitive type value, a wrapper class object or any library or user defined class object, provided it supplies a method `equals` for testing equality between two instances of the class itself. In particular, an `Lvar`-value can be an instance of `Lvar`, `Lst`, or `Set`.

Definition 3.2 *(Uninitialized logical variable) A logical variable which has no* `Lvar`*-value associated with it is said to be* uninitialized *(or an* unknown*). Otherwise, the logical variable is* initialized*. Also a logical variable whose* `Lvar`*-value is an uninitialized logical variable (or an uninitialized list or set) is said to be uninitialized.*

A value for a logical variable can be specified either when the variable is created or as the result of processing constraints involving the variable itself, in particular, equality constraints.

`Lvar` objects are manipulated basically through constraints (see Sect. 4). Constraints can be used to set the value of an uninitialized logical variable or to inspect it. No constraint is allowed to modify the value of a logical variable, by replacing it with a new one, i.e., logical variables are single-assignment variables [23].

Besides constraints, the class `Lvar` provides a number of utility methods that allow to read/write the value of a logical variable, to know whether a logical variable is initialized or not, to get the external name of a logical variable, and so on.

3.2 List and set objects

Definition 3.3 *(List) A* list *is a finite (possibly empty) sequence of arbitrary* `Lvar`*-values (the* elements *of the list). In JSetL a list is an instance of the class* `Lst`*, created by the declaration*

```
Lst LstName = new Lst(LstNameExt,LstElemValues);
```

where LstName *is the list name,* LstNameExt *is an optional external name, and* LstElemValues *is an optional part which is used to specify the elements of the list and can be either an array of elements $c_1, \ldots, c_n$ of any* `Lvar`*-value type, or the limits l and u of an interval $[l, u]$ of integer numbers or another list object. The* empty list *is denoted by the constant* `Lst.empty`.

Lists can be either initialized or uninitialized, like logical variables. The value of a list, i.e., a `Lst`-*value*, is the sequence of its elements. No typing information on elements of a list are provided, that is elements can be values of any type allowed for `Lvar` objects, including (initialized or uninitialized) logical variables, lists or sets.

Definition 3.4 *(Set) A* set *is a finite (possibly empty) collection of arbitrary* `Lvar`*-values (the* elements *of the set). In JSetL a set is an instance of the class* `Set`*, created by the declaration*

```
Set SetName = new Set(SetNameExt,SetElemValues);
```

where SetName*,* SetNameExt*, and* SetElemValues *have the same meaning as for lists. The constant* `Set.empty` *is used to denote the* empty set.

Like lists, sets can be either initialized or uninitialized, and no typing information are associated with set elements. The value of the set, i.e., a `Set`-*value*, can be specified when creating the set through the **new** operator in exactly the same ways seen for lists: by enumerating its elements in an array or, implicitly, by providing the limits of an interval of integer numbers, or by passing another set. Differences between lists and sets will become more evident when operating on them through list/set operations (e.g., list/set unification—see Sect. 4).

Example 3.5 `Lvar`*,* `Lst`*, and* `Set` *definitions*

```
Lvar x = new Lvar();             // an uninitialized l. var.
Lvar y = new Lvar("y",'a');      // an initialized l. var.  (value 'a'),
                                 // with ext'l name "y"
Lvar t = new Lvar(x);            // an uninitialized l. var.
                                 // (same as variable x)
Lst e = Lst.empty;               // the empty list
Lst l = new Lst("l");            // an uninitialized list,
                                 // with ext'l name "l"
Set i = new Set("i",4,4+3);      // an initialized set,
                                 // with ext'l name "i" and value {4,5,6,7}
int[] s_elems = {2,4,8,3};
Set s = new Set("s",s_elems);    // an initialized set (value {2,4,8,3}),
                                 // with ext'l name "s"
Lvar r = new Lvar(new Set());    // an uninitialized l. var,
                                 // whose value is an uninitialized set
```
□

Hereafter, we will often make use of an abstract notation—which closely resembles that of Prolog—to write lists and sets in a more convenient way. Specifically, $[e_1, e_2, \ldots, e_n]$ is used to denote the list containing n elements $e_1, e_2, \ldots, e_n$, while $[\,]$ is used to denote the *empty list*. Moreover, $[e_1, e_2, \ldots, e_n \mid R]$, where R is a list, is used to denote a list containing the n elements $e_1, e_2, \ldots, e_n$, plus elements in R. In particular, if R is uninitialized, $[e_1, e_2, \ldots, e_n \mid R]$ represents an "*unbounded*" list, with elements $e_1, \ldots, e_n$ and an unknown part R. Similar abstract notations are used also to represent sets (with square brackets replaced by curly brackets).

Note that we will use the term *list* (resp., *set*) to refer to both a `Lst` object (resp., a `Set` object), and a `Lvar` object whose value is a `Lst` (resp., a `Set`) object. Actually, in JSetL they are treated as completely equivalent objects as far as the possible values and operations that can be applied to them are concerned. The only difference is that using `Lst` (`Set`) objects allows in general a more controlled usage of the objects. For example, if m is a method not defined for `Lst`, trying to apply m to a list will be detected as an error at compile-time, whereas applying m to a logical variable initialized to a list will only cause an exception to be raised at run-time (namely, the exception `NotLstException`).

Elements of a list/set can be also logical variables (or lists or sets), possibly uninitialized.

Definition 3.6 *(Partially specified list/set) A list (resp., set) that contains some elements which are uninitialized is said to be a* partially specified *list (resp., set).*

Definition 3.7 *(Nested list/set) Elements of a list (resp., set) which are themselves lists (resp., sets) are said to be* nested lists *(resp.,* nested sets*). Lists/sets can be* nested *at any depth.*

Example 3.8 *Partially specified and nested lists/sets*

```
Lvar x = new Lvar();
Object[] pl_elems = {new Integer(1),x};
Lst pl = new Lst(pl_elems);              // the list [1,x]

Set[] ns_elems = {i,s,r};                // i, s, and r are the sets
                                         // defined in Example 3.5
Set ns = new Set(ns_elems);
```

`pl` *is a partially specified list, containing an* `Integer` *object and an uninitialized logical variable* `x`. `ns` *is a set containing three nested sets (either initialized or uninitialized).* □

Note that the cardinality of a (bounded) list is always known even if some of its elements are not specified, whereas the cardinality of a partially specified set is not completely determined. For example, the partially specified set `{1,x}` has cardinality 1 or 2 depending on whether `x` will get initialized to `1` or to a value different from `1`, respectively. Actually, each partially specified list/set denotes a possibly infinite collection of different lists/sets, that is all lists/sets which can be obtained by assigning admissible values to the uninitialized variables.

Sets are manipulated mainly through constraints (see Sect. 4). Conversely, the current version of the JSetL library does not support constraint over lists, apart from list equality and inequality.[1] Both lists and sets, however, can be manipulated also through a number of *utility methods* which are made available by classes `Lvar`, `Lst`, and `Set`. These methods are used, for example, to print a list/set object, to inspect it (e.g., to test if it is completely specified), to get the external name, and so on. Being implemented as usual methods (not as constraints), these operations require that their arguments are sufficiently instantiated to be processed.

In the next subsection, we describe in particular the element insertion and extraction methods for lists and sets, since they play an important role in the definition and manipulation of lists/sets. For other utilities methods refer to [19].

3.3 List and set constructor expressions

A list (resp., a set) can be also obtained as the result of evaluating a list (resp., a set) constructor expression.

Definition 3.9 *(List constructor) Let e be an* `Lvar` expression, *i.e. an expression returning any* ***`Lvar`****-value,* `l` *be a* list expression, *i.e., an expression returning a* ***`Lst`****-value, and* `x` *be an uninitialized logical variable. A* list constructor *is an expression of one of the forms:*

(*i*)	`l.ins1(`e`)`	*(head element insertion)*
(*ii*)	`l.insn(`e`)`	*(tail element insertion)*
(*iii*)	`l.ext1(x)`	*(head element removal)*
(*iv*)	`l.extn(x)`	*(tail element removal)*

The meaning of these expressions is as follows. Expressions (*i*) and (*ii*) denote the list obtained by adding the value returned by the evaluation of e (i.e., $val(e)$) as the first or the last element of the list `l`, respectively, whereas expressions (*iii*) and (*iv*) denote the list obtained by removing from `l` the first or the last element,

[1] This limitation will be possibly removed in future releases by integrating other basic constraints over lists with set constraints as proposed for instance in [11].

respectively. Evaluation of expressions (*iii*) and (*iv*) also causes the value of the removed element to become the value of `x`. [2]

Constructor expressions for sets are simpler than those for lists. In fact, in lists we can distinguish between the first (the *head*) and the last (the *tail*) element of a list, while in sets the order of elements is immaterial. Moreover, only the element insertion method is provided since element extraction may involve a nondeterministic selection of the element to be extracted that is better handled using set constraints (see Section 4).

Definition 3.10 *(Set constructor) Let e be an* `Lvar` expression*, and* `s` *be a* set expression*, i.e., an expression returning a* ***Set****-value. A* set constructor *is an expression of the form:*

`s.ins(`e`)` *(element insertion)*

The meaning of `s.ins(`e`)` is the set obtained by adding $val(e)$ to `s` (i.e., `s` $\cup \{val(e)\}$).

It is important to notice that all these methods are declarative. In fact they do not modify the list/set on which they are invoked: rather they build and return a new list/set obtained by adding/removing the elements to/from the given one.

Insertion and extraction methods can be concatenated, left associative. In fact these methods always return a `Set`/`Lst` object, and the returned object can be used as the invocation object as well.

Using the insertion methods it is possible to build *unbounded* partially specified lists/sets, that is data structures with a certain number of (either known or unknown) elements $e_1, ..., e_n$, and an unknown rest part, represented by an uninitialized list/set r (i.e., using the abstract notation, $\{e_1, \ldots, e_n \mid r\}$ and $[e_1, \ldots, e_n \mid r]$ for sets and lists, respectively).

Example 3.11 *List/Set element insertion and removal*

```
Lvar x = new Lvar();        // an uninitialized l. var.
Lst nil = Lst.empty;        // the empty list
Lst l1 = nil.ins1(3+2).ins1(x);
                            // the list [x,5]
Lvar y = new Lvar();        // an uninitialized l. var.
Lst l2 = l1.ext1(y).insn(y);
                            // the list [5,x]
Set s1 = Set.empty.ins(1).ins('a');
                            // the set {'a',1}
Set r = new Set();          // an uninitialized set
Set s2 = r.ins(1);          // the unbounded set {1|r}
```

□

Note that `s2` in the above example is an unbounded partially specified set containing one element, `1`, and an unknown part `r`; in this case, the cardinality of the denoted set has no upper bound (the lower being 1).

[2]Extraction methods for lists require that the invocation list `l` is initialized and that `x` is not initialized. If one of these conditions is not respected an exception is raised (namely, `NotInitVarException` and `InitLvarException`, respectively). Moreover, if `l` is the empty list, the exception `EmptyLstException` is raised.

Special forms of the insertion and extraction methods are provided to simplify their usage. In particular, the method `ins1All(a)`, applied to a list `l`, where `a` is an array of elements of a type t, returns a list obtained from `l` by adding all elements of `a` to `l`, respecting the order they have in `a`. Similarly, `insAll(a)`, applied to a set `s`, is used to insert more than one element at a time into `s`.

3.4 Programming examples

Logical variables, along with partially specified data structures, are fundamental notions for the declarative computation model (see, e.g., [23]). In this sub-section, we show a few simple examples where these notions are exploited to get more flexible, and possibly easier to write and understand, Java programs.

Example 3.12 *Translate the name of the days of the week from Italian to English and vice versa.*

```
static SolverClass Solver = new SolverClass();
public static void weekTranslate(Lvar g, Lvar d) {
   boolean match = false;
   if (Solver.boolSolve(g.eq("lunedi").and(d.eq("Monday"))) ||
       Solver.boolSolve(g.eq("martedi").and(d.eq("Tuesday"))) ||
       Solver.boolSolve(g.eq("mercoledi").and(d.eq("Wednesday"))) ||
       Solver.boolSolve(g.eq("giovedi").and(d.eq("Thursday"))) ||
       Solver.boolSolve(g.eq("venerdi").and(d.eq("Friday"))) ||
       Solver.boolSolve(g.eq("sabato").and(d.eq("Saturday"))) ||
       Solver.boolSolve(g.eq("domenica").and(d.eq("Sunday")))) match=true;
   if (match) return;
   else  Solver.failure();
}
```

The equality operator `eq` implements unification. Equalities, as well as other constraints on logical variables, can be solved immediately by invoking the method `boolSolve` over an instance of `SolverClass`. `Solver` is the `SolverClass` instance that will be used in examples throughout the paper.

The use of logical variables and unification, in place of assignment and standard equality, allows the same methods to be used for different purposes. For example, if `answer`, `lunedi`, and `monday` are logical variables declared as follows:

```
Lvar answer = new Lvar();
Lvar lunedi = new Lvar("", "lunedi");
Lvar monday = new Lvar("", "Monday");
```

the following three are all legal invocations of the method `weekTranslate` of Example 3.12:

```
weekTranslate(lunedi,answer);
weekTranslate(answer,monday);
weekTranslate(lunedi,monday);
```

That is, the same method can be used both to test a given translation (the third invocation) and to translate a given week name into another one (the first and second invocations). No assumption is made about which are input and which are output parameters. In the first invocation, the first parameter is an input and the second is an output, and vice versa in the second invocation, while in the third invocation both are input parameters. Actually, the method `weekTranslate` defines a binary *relation*, not a function that maps one or more inputs to exactly one output.[3]

If no matches are found for the passed values, `weekTranslate` calls the method `failure` of `Solver`. This simply causes the exception `Failure` to be raised. The default action for this exception is the immediate termination of the current thread. The exception, however, can be caught by the program and dealt with as preferred.

The real power of logical variables becomes more apparent when considering their usage in defining and handling partially specified data structures, as shown in the following two examples.

Example 3.13 *Check whether a list* `L` *contains at least two elements.*

```
public static void at_least_two(Lst L)
throws Failure {
    Lvar x = new Lvar();
    Lvar y = new Lvar();
    Lst R  = new Lst();                        // an uninitialized list
    Solver.solve(L.eq(R.ins1(y).ins1(x)));  // L = [x,y | R]
    return;
}
```

The problem is solved—in a truly declarative way—by requiring that the given list `L` satisfies the equality $\mathtt{L} = [\mathtt{x},\mathtt{y}|\mathtt{R}]$. Roughly speaking, the method `solve` differs from the method `boolSolve` in that the former does not return an explicit result but it causes the exception `Failure` to be generated if its argument turns out to be not satisfiable in the context of the current collection of constraints.

Example 3.14 *(Deterministic) list concatenation. Given two lists* `L1` *and* `L2` *computes the new list* `L3` *as the concatenation of* `L1` *and* `L2`*.*

```
public static void concat(Lst L1, Lst L2, Lst L3)
throws Failure {
    if (Solver.boolSolve(L1.eq(Lst.empty))) {
        Solver.solve(L2.eq(L3));  // L3 = L2
    }
    else {
        Lvar x     = new Lvar();
        Lst R      = new Lst();
        Lst L3new  = new Lst();
        Solver.solve(L1.eq(R.ins1(x)).and(L3.eq(L3new.ins1(x))));
                                  // L1 = [x|R] ∧ L3 = [x|L3new]
        concat(R,L2,L3new);
    }
    return;
}
```

[3]Note that the use of uninitialized logical variables as output parameters provides a simple solution to the problem caused by the fact that Java always passes primitive types and object references to methods by value.

Observe that, in this version of `concat`, `L1` and `L2` are required to be input parameters. `L3` instead can be either an output or an input parameter (i.e., `concat` can be used also to check whether a list is the concatenation of two given lists). A fully invertible definition of `concat` requires nondeterminism and will be shown in Sect. 5.

4 (Set) Constraints

Logical variables and sets are used mainly in conjunction with *constraints*. Constraints are handled by a *constraint solver*, that is an instance of the class `SolverClass`, which implements our specific constraint solving strategy. Constraints are passed to a constraint solver S adding them to its *constraint store*, which constitutes the current collection Γ of active constraints for S. Resolution of the current constraint Γ is basically performed by calling the method `solve` of the constraint solver S. The solver nondeterministically searches for a solution that satisfies all constraints introduced in the constraint store, by rewriting constraints to a simpler form. Those constraints, usually involving one or more uninitialized variables, which cannot be further simplified are maintained in the constraint store and will be used to narrow the set of possible values that can be assigned to variables occurring in them.

4.1 JSetL constraints

The JSetL *constraint domain* is the $\mathcal{SET}$ domain defined in [10], extended with a few new constraints for comparison over integers.

Definition 4.1 *(Atomic constraint) An* atomic constraint *in JSetL is an expression of one of the forms:*

- $e_1.op\,(e_2)$
- $e_1.op\,(e_2, e_3)$

*where op is one of a collection of predefined methods (*eq*,* neq*,* in*,* nin*,* subset*,* union*,* inters*,* disj*,* differ*,* nsubset*,* nunion*,* ninters*,* ndisj*,* ndiffer*,* less*,* le*,* ge*,* le*,* gt*) and e_1, e_2 and e_3 are expressions whose type depends on op.*

The meaning of these methods is that naturally associated with their names: eq and neq stand for equality and inequality; in and nin, for membership and not membership; le ($\leq$), ge ($\geq$), le ($<$), gt ($>$), for integer comparison relations; subset, union, inters, ..., along with their negative counterparts (e.g., nsubset), for basic set-theoretic operations.

As concerns expressions e_1, e_2, e_3, when they are used to denote sets in set-theoretic operations (e.g., e_2 in $e_1.\mathsf{in}\,(e_2)$), their type is restricted to be `Set` or `Lvar`. In the second case, whenever the `Lvar` object has got a value, a run-time check is performed to ensure that the value is of type `Set`; otherwise the exception `NotSetException` is raised. Similarly, expressions denoting integers in comparison operations (e.g., e_1 and e_2 in $e_1.\mathsf{ge}\,(e_2)$) are restricted to be of type `int` or `Integer` or `LvarExpr`. `LvarExpr` is a class representing an arithmetic expression possibly involving `Lvar` objects and built using the arithmetic methods `sum`, `sub`, `mul`, `div`, and `mod`. Also `Lvar` objects occurring in `LvarExpr` expressions, that must be instantiated to only integer values, are checked at run-time for type correctness.

Definition 4.2 *(Constraint) A* constraint *is either an atomic constraint or (recursively) an expression of the form:*

- c_1.and (c_2) ... .and (c_n)

where $c_1, c_2, \ldots, c_n$ are atomic constraints.

The meaning of c_1.and(c_2) ... .and(c_n) is the logical conjunction $\hat{c}_1 \wedge \hat{c}_2 \wedge \ldots \wedge \hat{c}_n$, where $\hat{c}_1 \ldots \hat{c}_n$ are the expressions representing the meaning of $c_1 \ldots c_n$, respectively. Hereafter, we will usually use the term *constraint* to refer to both the syntactic object that represents a constraint in JSetL (e.g., c_1.and (c_2) ... .and (c_n)) and the logical expression that denotes its meaning (e.g., $\hat{c}_1 \wedge \hat{c}_2 \wedge \ldots \wedge \hat{c}_n$). Moreover, with a little abuse of notation, we will often use the logical expression to represent the corresponding JSetL constraint. When useful, however, we can distinguish them by using the notations C and $\hat{C}$, for the JSetL constraint and the logical expression representing its meaning, respectively.

Example 4.3 *JSetL constraints*

Let `x`*,* `y`*,* `z` *be logical variables and* `r`*,* `s`*, and* `t` *be sets.*

```
r.eq(s);                    // equality between sets
t.union(r,s);               // t = r ∪ s
x.eq(y.sum(1)).and(x.eq(3)).and(y.neq(z))
                            // x = y + 1 ∧ x = 3 ∧ y ≠ z
```

□

Note that solving equality constraints in JSetL implies, in general, the ability to deal with *set unification* (cf., e.g., [12]). Set unification of two possibly partially specified sets s and r means finding an assignment of values to uninitialized variables occurring in them (if any) that makes them equal in the underlying set theory. Intuitively, in any reasonable set theory, two sets are equal if they have the same elements, disregarding their order and possible repetitions. Thus, for instance, the set unification problems

$$\{x, 2\} = \{y, 1\}$$
$$\{x, 2\} = \{1, 1, y\}$$

where x and y are uninitialized logical variables, are both satisfied by $x = 1$ and $y = 2$. In general, a set unification problem can admit more than one solution. For example, the equation

$$\{x, y\} = \{z, 2\}$$

is satisfied by x, y and z equal to 2 or by $x = z$ and $y = 2$ or by $y = z$ and $x = 2$. The constraint solver is able to compute, one after the other, all solutions for a given unification problem.

4.2 The Constraint Store

The constraint store of a constraint solver S contains the collection of all active constraints for S in the running program. JSetL provides methods through which the program can add new constraints to a specific constraint store, visualize its content, and remove all constraints stored in it.

The addition of a constraint to a constraint store is (basically) performed by calling the method add of the class SolverClass. The invocation

$$S\texttt{.add(}C\texttt{)}$$

adds the constraint C to the constraint store of the constraint solver S. The collection of constraints stored in a constraint store is interpreted as a conjunction of atomic constraints. If Γ is the constraint currently stored in the constraint store, the (meaning of the) constraint obtained by executing S.add(C) is $\hat{\Gamma} \wedge \hat{C}$. The order in which constraints are added to the constraint store is completely immaterial.

After constraints have been added to the constraint store, one can ask the constraint solver S to solve them, e.g. by invoking the method solve:

$$S\texttt{.solve().}$$

The method solve nondeterministically searches for a solution that satisfies all constraints introduced in the constraint store. Constraint solving will be addressed in detail in the next subsection.

Example 4.4 *Adding constraints*

Let x, y, *and* z *be uninitialized logical variables, and* Solver *be an instance of* SolverClass.

```
Solver.add(x.eq(3));     // equality constraint x = 3
Solver.add(y.neq(x));    // inequality constraint y ≠ x
Solver.add(x.eq(z));     // equality constraint x = z
```

The constraint added to the constraint store is $\texttt{x} = 3 \wedge \texttt{y} \neq \texttt{x} \wedge \texttt{x} = \texttt{z}$. *The same effect can be obtained by executing*

```
Solver.add(x.eq(3).and(y.neq(x)).and(x.eq(z)));
```

Clearly, the constraint is satisfiable. After the execution of the method solve, *the logical variables* x *and* z *will have both value* 3, *while* y *will remain uninitialized.* □

At each moment, the constraint store of a solver S is constituted by all atomic constraints that have been added to it and not yet processed, as well as by the *irreducible* (or solved form) constraints that have been processed by the solver and that cannot be further simplified (see next subsection). All constraints stored in the constraint store can be visualized at any moment using the method showStore.

JSetL provides also a more convenient way to introduce more than one constraint at a time, through the method forall. Let x be an uninitialized logical variable, s a set expression, C a constraint containing x. The statement

$$S\texttt{.forall(x,}s\texttt{,}C\texttt{)}$$

adds to the constraint store of S, for each element e of s, an instance of C where each occurrence of x in C is replaced by e. In more abstract terms, the sequence S.forall(x,s,C); S.solve() corresponds to the so-called *Restricted Universal Quantifier* (cf., e.g., [10]), $(\forall x \in s)C$, whose logical meaning is

$$\forall x((x \in s) \rightarrow C).$$

Example 4.5 *Simple use of* forall

```
Lvar x = new Lvar();           // an uninitialized l. var.
int[] s_elems = {2,8,5};
Set s = new Set(s_elems);      // the set {2,8,5}
Solver.forall(x,s,x.ge(0));
```

The constraint added to the constraint store of `Solver` *is* $2 \geq 0 \wedge 8 \geq 0 \wedge 5 \geq 0$ *(clearly satisfiable).* □

The set s in an invocation `Solver.forall(x,`s`,`C`)` can be also uninitialized or, if initialized, it can be either bound or unbound, and in both cases it can contain uninitialized logical variables as its elements. In all these cases, the subsequent call of the constraint solver will nondeterministically compute all possible values for the uninitialized variables occurring in s such that the conjunction of constraints added to the constraint store is satisfied. Hence, the method `forall` can be used not only to "iterate" over all elements of a given set, but also to *generate* the set, or part of the set, whose elements satisfy a given property. For example, if `s` is the set `{x1,x2}`, with `x1` and `x2` uninitialized logical variables, and `r` is the set `{1,2}`, executing

`Solver.forall(x,s,x.in(r))`

causes the constraint `x1` $\in$ `{1,2}` $\wedge$ `x2` $\in$ `{1,2}` to be added to the constraint store, which holds for either `x1 = 1` and `x2 = 2`, or `x1 = 2` and `x2 = 1`, or `x1 = x2 = 1`, or `x1 = x2 = 2`.

The constraint C in `Solver.forall(x,`s`,`C`)` can contain also other uninitialized logical variables y_1, y_2,...,y_n, in addition to `x`. In this case, the instances of C which are added to the constraint store all share the same variables y_1, y_2,...,y_n. There are situations, however, in which one wants to have different instances of the variables y_1, y_2,...,y_n for each different instance of the constraint C added to the constraint store. Logically, this corresponds to consider variables y_1, y_2,...,y_n in C as existentially quantified within the scope of the Restricted Universal Quantifier, that is, $\forall x((x \in s) \rightarrow \exists y_1, \ldots, y_n(C))$. For this purpose, JSetL provides also the method

`Solver.forall(x,`s`,`Y`,`C`)`

where `x`, s, and C are the same as in the simpler method `forall`, while Y is an array of all the *local* uninitialized logical variables $y_1, \ldots, y_n$ occurring in C. Concretely, a new copy of these variables is created for each element `x` of the set `s`.

As an example to illustrate this more elaborated form of the method `forall`, let us consider the following problem.

Example 4.6 *(*`forall` *with local variables). Check whether all elements of a set* `s` *are pairs, i.e., they have the form* `[y1,y2]`*, for any* `y1` *and* `y2`*.*

```
Lvar x = new Lvar();                      // an uninitialized l. var.
Lvar y1 = new Lvar();                     // an uninitialized l. var.
Lvar y2 = new Lvar();                     // an uninitialized l. var.
Lvar[] Y = {y1,y2};                       // an array of two l. var's.
Lst pair = Lst.empty.ins1(y1).insn(y2);   // the list [y1,y2]
Solver.forall(x,s,Y,x.eq(pair));
Solver.solve();                           // ∀x(x ∈ s → ∃y1,y2 x = [y1,y2])
```

Let `s` *be the set* `{[1,3],[1,2],[2,3]}`*. The constraint added to the constraint store is:* $[1,3] = [\mathtt{y1}_1, \mathtt{y2}_1] \wedge [1,2] = [\mathtt{y1}_2, \mathtt{y2}_2] \wedge [2,3] = [\mathtt{y1}_3, \mathtt{y2}_3]$ *which is clearly* *satisfiable. Note that, if* `y1` *and* `y2` *were not defined as "local" to the invocation of the method* `forall` *(i.e., the* `Y` *parameter is omitted), all equality constraints added to the constraint store would have the same variables* `y1`, `y2` *and the whole constraint would turn out to be unsatisfiable.* □

4.3 Constraint solving

The approach adopted for constraint solving in JSetL is the same developed for CLP($\mathcal{SET}$) [10]. The CLP($\mathcal{SET}$) constraint solver tries to reduce any conjunction of atomic constraints to a simplified form - called the *solved form* - which is proved to be satisfiable. The success of this reduction allows one to conclude the satisfiability of the original collection of constraints. On the contrary, the detection of a failure (logically, the reduction to false) implies the unsatisfiability of the original constraints.

Definition 4.7 *(Solved form) A constraint C is in* solved form *if it is empty or all its constituting atomic constraints have one of the following forms (where x, x_i are uninitialized logical variables or sets, e is an* ***Lvar*** *expression, o is either an* `Lvar` *or a* `List` *or a* `Set` *expression—cf. Definition 3.9):*

1. x`.eq(`e`)`*, and x does not occur neither in e, nor in the other constraints of C;*
2. x`.neq(`e`)`*, and x does not occur in e;*
3. o`.nin(`x`)`*, and x does not occur in o;*
4. x_1`.disj(`x_2`)`*, and $x_1 \not\equiv x_2$;*
5. x_3`.union(`x_1`,`x_2`)`*, with $x_1 \not\equiv x_2$ and there are no constraints of the form* x_1`.neq(`e`)`*, or* x_2`.neq(`e`)`*, or* x_3`.neq(`e`)` *in C.*

A solved form constraint obtained from a constraint C represents a *solution* for C. A constraint C can have more than one solution. The JSetL constraint solver is able to find nondeterministically all correct solutions for a given constraint. Nondeterminism is handled through *choice points* and *backtracking*. Once the constraint reduction process detects a failure, the computation backtracks to the most recently created choice point (chronological backtracking). If no choice point is left open, the whole reduction process fails and the exception `Failure` is generated.

Definition 4.8 *(Failure) We say that the computation* terminates with failure *if it causes the exception* `Failure` *to be raised. Otherwise, the computation* terminates with success.

Solved form constraints are *irreducible* constraints. As such, they are left in the constraint store and possibly passed ahead to a new invocation of the constraint solver. A successful computation, therefore, can terminate with a not empty collection of solved form constraints in the final constraint store. Conversely, constraints not in solved form are always eliminated by the constraint solving process: in fact, they are reduced to solved form constraints or rewritten to false, or they cause an exception to be raised.

The current version of the JSetL constraint solver is not able to deal with integer comparison constraints in a complete way. Specifically, integer constraints containing uninitialized logical variables are simply delayed as much as possible, leaving them unresolved in the constraint store. As soon as we need to check constraint satisfiability, if the solver detects that the constraint store still contains integer constraints with uninitialized logical variables then the exception `Uninitialized_Variable_in_arithmetical_expression` is raised. In other words, in these cases the constraint solver notifies the caller of the fact that it has not enough information to decide satisfiability of the given constraints.

The JSetL constraint solver is basically a Java implementation of the CLP($\mathcal{SET}$) solver [10], extended with simple constraints over integers. The treatment of integer constraints, however, does not affect its soundness and termination properties which are therefore guaranteed to hold as in the original proposal. On the other hand, completeness is only guaranteed if the constraint does not contain integer constraints with uninitialized logical variables.[4]

The JSetL constraint solver is implemented by the class `SolverClass`. The main method for constraint solving is the method `solve`. If `Solver` is an instance of `SolverClass`, `Solver.solve()` either detects that the constraint in the constraint store of `Solver` is unsatisfiable and raises the exception `Failure`, or it transforms nondeterministically the constraint stored in the constraint store into a simplified form.

Example 4.9 *(Simple constraint solving). Check whether an element* `x` *belongs to the difference between two sets,* `s1` *and* `s2` *(i.e.,* $x \in s1 \setminus s2$*).*

```
public static void in_difference(Lvar x, Set s1, Set s2)
throws Failure {
    Solver.add(x.in(s1));
    Solver.add(x.nin(s2));
    Solver.solve();
    }
```

If the following code fragment is executed (for instance, in the method **main***)*

```
in_difference(x,s,r);
x.output();
Solver.showStore();
```

and `s` *and* `r` *are the sets* `{1,2}` *and* `{1,3}`*, respectively, and* `x` *is an uninitialized variable, the output generated is:*

```
x = 2
Store:  empty
```

Conversely, if `s` *is an uninitialized set, then executing the same program fragment as above, will produce the following output*

```
x = unknown
s = {x | Set_1}
Store:  x.neq(1) x.neq(3)
```

which is read as: `s` *can be any set containing the element* `x` *and* `x` *must be different from* `1` *and* `3`*.* □

[4]These limitations will be possibly removed in future releases, following, for example, the approach described in [7], where the $\mathcal{SET}$ solver is integrated with an efficient solver over *finite domains* which allows the language to deal with arithmetic constraints with almost no restriction on the instantiation of expressions that occur in them.

The simplified constraint returned by the method `solve` is not necessarily a solved form constraint but a weak variant of it.

Definition 4.10 *(Weak solved form) A constraint C is in* weak solved form *if it empty or all its constituting atomic constraints have either one of the forms (1)—(4) of Definition 4.7 or one of the additional forms:*

- *an atomic comparison constraint containing uninitialized logical variables;*
- x_3`.union(`x_1`,`x_2`)`, *with $x_1 \not\equiv x_2$ (i.e., the conditions over* `union` *constraints of the definition of the solved form are relaxed).*

The weak solved form is not guaranteed to be satisfiable in general.

Example 4.11 *Unsatisfiable weak solved forms*

If the following sequence of statements is executed

```
Solver.add(x.le(y).and(x.gt(y)));
Solver.solve();
```

with `x` *and* `y` *uninitialized logical variables, the resulting constraint store will contain the weak solved form constraint* $\mathtt{x} < \mathtt{y} \wedge \mathtt{y} > \mathtt{x}$ *which is clearly unsatisfiable.*

Analogously, the following sequence of statements

```
Solver.add(x.subset(y).and(y.subset(x)).and(x.neq(y)));
Solver.solve();
```

with `x` *and* `y` *uninitialized sets, causes the constraint solver to produce the constraint* $\mathtt{y} = \mathtt{x} \cup \mathtt{y} \wedge \mathtt{x} = \mathtt{y} \cup \mathtt{x} \wedge \mathtt{x} \neq \mathtt{y}$ *which is in weak solved form but it is clearly unsatisfiable.* □

We can now give a more precise definition of the invocation of the methods `solve` and `boolSolve`.

Definition 4.12 *(*`solve` *and* `boolSolve`*) Let* `S` *be the constraint solver, Γ the constraint currently stored in the constraint store of* `S`*, and C a constraint.*
The invocation `S.solve()` *(resp.,* `S.solve(`C`)`*) terminates with success if the constraint Γ (resp., Γ.and (C)) can be reduced to a weak solved form; otherwise, the invocation terminates with failure.*
The invocation `S.boolSolve()` *(resp.,* `S.boolSolve(`C`)`*) always terminates with success and it returns* true *if the invocation* `S.solve()` *(resp.,* `S.solve(C)`*) terminates with success, otherwise it returns* false.[5]

If a constraint in weak solved form does not contain neither unresolved comparison constraints nor union constraints, then it is in solved form and as such it is guaranteed to be satisfiable. In general, to be sure to obtain only solved form constraints one must use the method `finalSolve`. This method actually invokes the method `solve` but it performs also a final global check on the generated constraint. If the constraint turns out to be unsatisfiable then the exception `Failure` is raised. If, on the contrary, the constraint solver detects that the constraint store still contains comparison constraints then the exception `Uninitialized_Variable_in_arithmetical_expression` is raised.

[5]Actually, `boolSolve` may raise an exception `Failure` but it also takes care of catching it and transforming it to a boolean result.

Definition 4.13 *(*`finalSolve`*) Let* `S`*,* Γ*, and* C *be as in Definition 4.12. The invocation* `S.finalSolve()` *(resp.,* `S.finalSolve(`C`)`*) terminates with success if the constraint* Γ *(resp.,* Γ.and (C)*) can be reduced to a solved form; otherwise, the invocation terminates with failure.*

As an example, using the method `finalSolve` (instead of the method `solve`) in the program fragments of Example 4.11, we get the exception `Uninitialized_Variable_in_arithmetical_expression` for the first call, and a failure (hence, the exception `Failure`) for the second one.

Constraints can be accumulated in the constraint store and then solved as a whole (using either `solve` or `finalSolve`), or they can be solved as they are added to the constraint store. As a matter of fact, if `S` is the constraint solver, and $C_1,\ldots,C_n$ are constraints, executing the sequence of statements

`S.add(`C_1`); S.add(`C_2`); ...S.add(`C_n`); S.solve();`

is equivalent, as far as constraint satisfiability is concerned, to execute

`S.add(`C_1`); S.solve(); S.add(`C_2`); S.solve();...; S.add(`C_n`); S.solve();`

Constraint solving in JSetL is *incremental.* Every time the method `solve` (or `finalSolve`) is invoked it does not restart solving the constraint from the beginning but it restarts from the point reached by the last invocation of `solve`. Satisfiability, however, is always checked with respect to the global constraint, stored in the constraint store during the whole computation. In particular, the choice points left open by a previous call to the solver are still open and they are explored by the new invocation if needed.

The method `finalSolve` can be used in place of the method `solve` or it can be used in conjunction with it. In the second case, the method `finalSolve` is usually called at the end of the constraint solving process, after one or more invocations of the method `solve`. This technique can be useful, on the one hand, to gain execution efficiency since the global check required by the `finalSolve` is performed only once. On the other hand, it can be used as a way to delay comparison constraints that are not enough specified: these constraints are simply postponed by the method `solve`, until the method `finalSolve` is invoked.

Besides the methods `solve`, `boolSolve`, and `finalSolve`, `SolverClass` supplies a few other methods for constraint solving: `solve1`, `nextSolution`, `setof`. Since these methods are tightly related to nondeterminism, we postpone their presentation to the next section.

4.4 Programming with constraints

The ability to solve constraints disregarding the fact logical variables occurring in them are initialized or not (i.e., to compute with partially specified values) has a number of potential advantages.

First of all, the ability to (re-)use the same methods for different purposes, that we have already pointed out for methods involving unification (see Sect. 3.4), can be generalized to methods using any constraints.[6] For example, the method

[6]Actually, unification is just a special case of constraint solving, where constraints are limited to equalities.

`in_difference` shown in Example 4.9, which uses membership and non-membership constraints, can be used either to test or to compute any of its arguments `x`, `s1`, `s2` such that `x` $\in$ `s1\s2` holds.

Another potential advantage of programming with constraints, which may contribute to support a declarative reading of a program, is the possibility to ignore (in some cases) the order of statement execution. For example, if `x` is an uninitialized logical variable, executing

```
x.read();
Solver.add(x.neq(0));
Solver.solve();
```

is the same as executing

```
Solver.add(x.neq(0));
x.read();
Solver.solve();
```

In addition to the above mentioned advantages, programming with constraints provides also a powerful modeling tool, particularly well suited to concisely express solutions for complex problems such as the constraint-satisfaction problems (CSP). As a matter of fact, constraint handling facilities allow the programmer to express (declarative) solutions to his/her problems as collections of equalities and inequalities, to be solved in a subsequent moment, with no concern to the order in which they occur in the program. That is, they allow the programmer to focus on *what* the program does, rather than on *how* it does.

As two further more comprehensive examples of the programming style supported by JSetL, using constraints as well as the other previously introduced facilities, such as logical variables, lists, and sets, we show the solution of two well-known CSP problems, the map coloring and the n-queens problems. Using JSetL allows solutions to these problems to be specified in a quite abstract and concise way, similar to what is obtained, for instance, by using an high-level executable specification language such as NP-SPEC [3].

Example 4.14 *Coloring of a map*

Given a map of n regions $r_1,\ldots,r_n$ and a set of m colors $c_1,\ldots c_m$ find an assignment of colors to regions such that neighboring regions have different colors.

*The regions are represented by a set of n uninitialized logical variables and the colors by a set of m constant values (e.g., {`"red","blue"`}). The map is modeled by an undirected graph and it is represented as a set whose elements are themselves sets containing two neighboring regions. At the end of the computation each **Lvar** representing a region will be initialized with one of the given color.*

```
public static void coloring(Set regions, Set map, Set colors)
throws Failure {
    Lvar x = new Lvar();
    Set single = Set.empty.ins(x);    //a singleton set
    Solver.add(regions.eq(colors));  //unify regions with colors
    Solver.forall(x,colors,single.nin(map));
    Solver.solve();
    return;
    }
```

The solution uses a pure "generate & test" approach. The `regions` = `colors` *constraint allows us to find a valuable assignment of colors to regions. Invocation of the method* `forall` *allows us to test whether the constraint* {`x`} $\notin$ `map` *holds for all* `x` *belonging to* `colors`*. If it holds, it means that for no pair* $\{r_i, r_j\}$ *in* `map`*,* r_i *and* r_j *have got the same color* `x`*.*

If `coloring` *is called with* `regions` = {`r1,r2,r3`}*,* `r1`*,* `r2`*,* `r3` *uninitialized logical variables,* `map` = {{`r1,r2`},{`r2,r3`}}*, and* `colors` = {`"red", "blue"`}*, the invocation terminates with success, and* `r1`*,* `r2`*,* `r3` *are initialized to* `"red"`*,* `"blue"`*, and* `"red"`*, respectively (actually, also the other solution which initializes* `r1`*,* `r2`*,* `r3` *to* `"blue"`*,* `"red"`*, and* `"blue"`*, respectively, can be computed through backtracking).*

It is important to note that this solution allows the set of colors to be also partially specified. For example, if `colors` = {`c1,"blue"`}*, with* `c1` *an uninitialized variable, executing* `coloring` *will generate the constraint:*

`r1` = `Lvar_1`, `r2` = `blue`, `r3` = `Lvar_1`, `Lvar_1.neq(blue)`

that is, regions `r1` *and* `r3` *can get any color different from* `blue`*.* □

Example 4.15 *The n-queens problem*

The n-queens problem consists in trying to place n queens on a $n \times n$ *chess board such that no two queens attack each other, i.e., no two queens are placed on the same row, the same column or the same diagonal.*

To model the problem we observe that any solution will have a queen on each row and one on each column. We can then represent each row with a logical variable, $r_0, ..., r_{n-1}$*. For every row there are n columns on which we can place the queen. If we identify the columns with the integer* $0, ..., n-1$*, the resolution of the problem consists in assigning a value among* 0 *and* $n-1$ *to each of the n logical variables* $r_0, ..., r_{n-1}$ *such that*

(*i*) $r_i \neq r_j$
(*ii*) $r_j + j - r_i \neq i$
(*iii*) $r_i + j - r_j \neq i$

for each $i \neq j$*,* $i \geq 0, j \leq n-1$*. The first inequality states that two queens can not be placed on the same column, while the other two inequalities state that two queens can not be placed on the same diagonal.*

The following is a complete Java program using JSetL for the n-queens problem, with $n = 4$*.*

```
import JSetL.*;
import java.io.*;

class Queens {
  static final int n = 4;
  static SolverClass Solver = new SolverClass;

  public static void main (String[] args)
  throws IOException, Failure {
    Set colmns = new Set(0, n-1); // a set of n integers representing columns
    Lst rows   = Lst.mkLst(n);     // a list of n Lvar's representing rows

    for (int i = 0; i < n; i++)     // assign a distinct number to each var. in rows
      Solver.add(((Lvar)rows.get(i)).in(colmns));
```

```
    Solver.allDifferent(rows);

    for (int i = 0; i < n-1; i++)  // add constraints on diagonals
      for (int j = i+1; j < n; j++) {
        Solver.add(((Lvar)rows.get(j)).sum(j).sub((Lvar)rows.get(i)).neq(i));
        Solver.add(((Lvar)rows.get(i)).sum(j).sub((Lvar)rows.get(j)).neq(i));
    }
    Solver.solve(); // check constraints

    for (int i = 0; i < n; i++) {  // print a solution
      System.out.print("\n Queen "+i+" = ");
      ((Lvar)rows.get(i)).print();
    }
    return;
  }
}
```

`get` *is an utility method for lists:* `l.get(i)` *returns the* `i`*-th element of the list* `l`*.* `allDifferent` *is used to add constraints to the constraint store: the invocation* `Solver.allDifferent(a)`*, where* `a` *is either a (bounded) list or set containing n elements $e_1,\ldots,e_n$, adds the atomic constraints $e_i \neq e_j$ to the constraint store of* `Solver`*, for all i, j in $1,\ldots,n$.* `sum` *and* `sub` *are methods of the class* `Lvar` *which implement addition and subtraction over integers, respectively: they allow to write usual arithmetic expressions but involving logical variables.*

Integers in set `colmns` *indicate the columns on which the queens can be placed. Logical variables in list* `rows` *indicate the rows of the chess board. A value c in the i-th* `Lvar` *of* `rows` *indicates a queen placed in row i at column c. The method* `allDifferent` *assures that only one queen is placed on each different column. The other two constraints added to the constraint store through the method* `add` *implement inequalities (ii) and (iii) (note that* `rows.get(i)` *and* `rows.get(j)` *denote the i-th and the j-th* `Lvar` *of the list* `rows`*, respectively).*

A possible solution printed by this program is:

```
Queen0 = 1
Queen1 = 3
Queen2 = 0
Queen3 = 2
```

that represents the situation depicted below

	$\mathcal{Q}$		
			$\mathcal{Q}$
$\mathcal{Q}$			
		$\mathcal{Q}$	

□

Remark 4.16 *The constraint solving technique supported at present by our constraint solver is basically a "generate & test". Solutions of the two CSP considered above use this technique, and therefore their execution time soon becomes unacceptably high as the complexity of data grows. As already mentioned, to overcome limitations of this approach—which are the same present in CLP($\mathcal{SET}$)—one could move* *along the lines suggested in [7]: integrating the current (set) constraint solver with a more efficient constraint solver over* finite domains *(FD). Whenever the JSetL solver detects constraints that can be handled as FD constraints (e.g., membership constraints over finite sets of integer numbers) it delegates their resolution to the FD solver. This allows us to maintain the expressive power and flexibility of the JSetL constraint domain in the general case, while permitting efficient FD constraint processing in many particular cases. In particular, solutions for the two sample problems discussed above could be formulated in almost the same way (the only requirement is to use integer numbers to represent colors in the map coloring problem) but their solution would be carried out much more efficiently by the FD solver embedded in the JSetL solver. These extension, however, is left for future work.*

5 Nondeterminism

A computation in JSetL can be nondeterministic, though nondeterminism in JSetL is confined to constraint solving. Precisely, like in SINGLETON [21], nondeterminism is mainly supported by set operations. As a matter of fact, the notion of nondeterminism fits into that of set very naturally. Set unification and many other set operations are inherently and naturally nondeterministic. For example, solving the set unification problem {`x,y,z`}={`1,2,3`} nondeterministically returns, one after the other, six different solutions:

```
x = 1, y = 2, z = 3,
x = 1, y = 3, z = 2,
x = 2, y = 3, z = 1, ...
```

and so on. Similarly, solving the membership problem $x \in \{1,2,3\}$, with x an uninitialized variable, nondeterministically returns one among $x = 1$, or $x = 2$, or $x = 3$.

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 (not trivial) notion of nondeterminism easier to understand and to use.

Nondeterminism provides a very high-level control abstraction. The ability to state solutions in a nondeterministic way, along with the availability of logical variables, represents another valuable support to obtain declarative programs.

We have already shown how to exploit this feature in the definition of the method `max` in Sect. 2.2 (using nondeterministic membership to select an element in the set) and in the solution of the coloring problem (using nondeterministic unification to find a possible coloring of regions). As a further example we show here a simple (declarative) solution to the common problem of computing permutations of a given set of numbers.

Example 5.1 *Compute all permutations (one at a time) of the set* {`1,2,3,4`}.[7]

[7] For the sake of simplicity we assume the problem deals with a specific set of integers. However, the solution could be easily generalized to any set: the cardinality `n` of the set can be computed using the method `size` of the class `Set`, while the set of `n` uninitialized variables to be used in the set unification problem can be constructed calling the method `mkSet(n)` of the class `Set`.

```
public static void main (String[] args)
throws IOException, Failure {
    int[] I_elems = {1,2,3,4};
    Set I = new Set(I_elems);   // I = {1,2,3,4}
    Lvar[] S_elems = {new Lvar(),new Lvar(),new Lvar(),new Lvar()};
    Set S = new Set(S_elems);   // S = {_S1,_S2,_S3,_S4}: a set of
                                // four anonymous variables
    Solver.solve(S.eq(I));      // {_S1,_S2,_S3,_S4} = {1,2,3,4}
    S.output();

    System.out.print("Do you want another solution(y/n)?");
    InputStreamReader reader = new InputStreamReader(System.in);
    BufferedReader input     = new BufferedReader(reader);

    while (input.readLine().equals("y")) {
        if (Solver.nextSolution()) {
            S.output();
            System.out.print("Do you want another solution(y/n)?");
        }
        else {
            System.out.println("no");
            break;
        }
    }
}
```

In this example we show also how to implement interaction with the user that allows him/her to get one solution at a time, when requested (a'la Prolog). The method `nextSolution` *allows one to obtain a new solution, if any, for the last issued* `solve` *(thus, in the above example, a new solution for the equality* `S.eq(I)`*). If no further solution exists,* `nextSolution` *returns a* false *result.* □

A simple way to exploit nondeterminism in JSetL is through the use of the `setof` methods of the class `SolverClass`. If `S` is the constraint solver, Γ is the constraint currently stored in the constraint store of `S`, and C is a constraint, the invocation

$$S\texttt{.setof(x,}C\texttt{)} \text{ (resp., } S\texttt{.setof(x)}\text{)}$$

collects into a set all values of the logical variable `x` satisfying the constraint $\hat{\Gamma} \wedge \hat{C}$ (resp., $\hat{\Gamma}$) and returns the collected set as its result.

Example 5.2 *Powerset*

Compute the set of all subsets (i.e., the powerset) of a given set `s`*.*

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

If `s` *is the set* `{'a','b'}`*, the set returned by* `powerset` *is* `{{}, {'a'}, {'b'}, {'a','b'}}`*.* □

The use of the `setof` methods can be seen as a way to provide a (limited) form of *intensional set definition* in JSetL. The intensional definition of a set is a very common way to represent a set: a set s is defined by providing a condition φ that is necessary and sufficient for an element x to belong to the set itself, i.e., $s = \{ x : \varphi \}$. Intensional set definitions represent another powerful tool for supporting a declarative programming style, as shown for instance in [10].

As an example, the following code defines, in a truly declarative way, a method to compute and print the set of all pairs `[x,y]` such that `x` and `y` belong to a given `s` and $\mathtt{x} \neq \mathtt{y}$, that is the set $\{ [\mathtt{x},\mathtt{y}] : \mathtt{x} \in \mathtt{s} \wedge \mathtt{y} \in \mathtt{s} \wedge \mathtt{x} \neq \mathtt{y} \}$.

Example 5.3 *All pairs*

```
public static Lvar allPairs(Set s)
throws Failure {
    Lvar x    = new Lvar();
    Lvar y    = new Lvar();
    Lst pair  = new Lst(Lst.empty.ins1(x).insn(y));  // a pair [x,y]
    Set Pairs = new Set();
    Pairs = Solver.setof(pair,x.in(s).and(y.in(s)).and(x.neq(y)));
                    // collect all pairs [x,y]
                    // satisfying the constraint x ∈ s ∧ y ∈ s ∧ x ≠ y
    Pairs.output();
    return;
}
```

If `s` *is the set* $\{$`1,2,3,4`$\}$ *(and the constraint store is initially empty), the set printed by executing this method is*

$\{$`[1,2],[1,3],[1,4],[2,1],[2,3],[2,4],[3,1],[3,2],[3,4],[4,1],[4,2],[4,3]`$\}$.

□

Observe that, the logical meaning of the above constraint is:

$$\forall \mathtt{pair}((\mathtt{pair} \in \mathtt{s}) \leftrightarrow \exists \mathtt{x},\mathtt{y}(\mathtt{x} \in \mathtt{s} \wedge \mathtt{y} \in \mathtt{s} \wedge \mathtt{x} \neq \mathtt{y})).$$

That is, the uninitialized logical variables possibly occurring in the constraint are considered as existentially quantified within the scope of the intensional definition of the set itself.

In the next example we make use of the other form of the method `setof` where the constraint is implicitly assumed to be the one currently stored in the constraint store. This form turns out to be particularly convenient when the constraint to be dealt with is quite complex.

Example 5.4 *Prime numbers*

Compute the set of all prime numbers smaller or equal to a given integer `n`*.*

```
public static Set primes(Lvar n)
throws Failure {
    Lvar x = new Lvar();
    Solver.add(x.in(new Set(2,n)));    // x ∈ 2..n
// check whether x is prime
    Lvar y = new Lvar();
    Lvar m = new Lvar();
```

```
      Solver.add(m.eq(x.sub(1)));          // m = x - 1
      Solver.forall(y,new Set(2,m),(x.mod(y)).neq(0));
   // compute the set of all primes numbers in 2..n
      return Solver.setof(x);
   }
```

In more abstract terms, the method `setof` *collects the set* $\{\mathtt{x} : \mathtt{x} \in \mathtt{2..n} \wedge \forall \mathtt{y}\ (\mathtt{y} \in \mathtt{2..x\text{-}1})\ \mathtt{x\ mod\ y} \neq \mathtt{0}\}$. □

Sometimes we are interested in computing just the first solution of a problem, not all solutions. In this cases one can use the method `solve1` instead of `solve`. `solve1` has exactly the same behavior of `solve` as concerns the first solution, but it does not keep track of the possible unexplored alternatives. In case of backtracking, these alternatives are simply ignored. For example, if we want to iterate over all elements of a set, by extracting one element at a time, and we are not concerned with the order of element extraction, using `solve1` instead of `solve` may lead to better execution efficiency, thanks to the pruning of the search tree. Clearly, `solve1` must be used with care, since it can cause useful solutions to be lost.

6 Program computation

JSetL is completely written in Java. As such, programs using JSetL can be understood as (usual) Java programs. However, those pieces of code, in particular those methods, that are written using only the "pure" declarative programming facilities provided by JSetL (e.g., without any destructive assignment), admit also a more abstract, declarative reading. Basically, the declarative computational model underlying programs using JSetL is that of the constraint logic programming with sets CLP($\mathcal{SET}$). As a matter of fact, "pure" declarative methods, such as the method `max` of Section 2.2, can be easily translated to a CLP($\mathcal{SET}$) program.

To understand the behaviour of the JSetL facilities in terms of this declarative computational model it is important to precisely specify the notion of a computation terminating with success or with failure.

The notion of failure has been introduced in Definition 4.8. The methods `solve` (as well as `solve1`) and `finalSolve` can terminate with either success or failure according to Definitions 4.12 and 4.13. `boolSolve` and `setof` always terminate with success. Conversely, the invocation `failure` always terminates with failure.

More generally, the invocation of a method or the execution of a statement may terminate with either success or failure according to the following definition.

Definition 6.1 *(Program computation) The invocation of a method or the execution of a statement that does not contain any (direct or indirect) invocation to one of the methods* `solve`*,* `solve1`*,* `finalSolve` *and* `failure` *always terminates with success. The statement* `if (`E`)` $S1$`; else` $S2$ *terminates with success if* E *is true and* $S1$ *terminates with success, or* E *is false and* $S2$ *terminates with success. The statement* `while (`E`)` S`;` *terminates with success if* E *is true and* S *terminates with success. A sequence of statements terminates with success if all its statements terminate with success. An invocation of a method terminates with success if the sequence of statements in the body of the method terminates with success. The* *program computation terminates with success if the main method terminates with success. In all other cases, the computation terminates with failure.*

The definition can be easily extended to other statements (e.g., `switch`, `for`, etc.).

With these definitions, along with the definitions for the methods `add`, `forall`, `setof`, etc. given in the previous sections, most of the examples presented in the previous sections admit a straightforward declarative reading. For example, the method `max` of Section 2.2 can be (informally) read as: "`max(s)` terminates with success returning `x` if there exists an `x` such that $\mathtt{x} \in \mathtt{s}$ and $(\forall \mathtt{y} \in \mathtt{s})\ \mathtt{y} \leq \mathtt{x}$ holds". Similarly, the method `concat` of Example 3.14 can be (informally) read as: "`concat(L1,L2,L3)` terminates with success if there exists a list `L3` such that `L1` is empty and $\mathtt{L3} = \mathtt{L1}$ holds, or `L1` is not empty and there exist `x`, `R`, `L3new` such that $\mathtt{L1} = [\mathtt{x}|\mathtt{R}] \wedge \mathtt{L3} = [\mathtt{x}|\mathtt{L3new}]$ holds and `concat(R,L2,L3new)` terminates with success".

7 Defining new constraints

JSetL allows the user to define new constraints. User-defined constraints are dealt with as the built-in constraints: they can be added to the constraint store using the method `add` and solved using the `SolverClass` methods for constraint solving. Definition of user-defined constraints, however, requires some programming conventions to be respected. In particular, all definitions must be provided as part of a class named `NewConstraints`, which is defined as belonging to the JSetL package (see [19] for a detailed description).

A possible usage of the new constraint facility is to allow part of a program to be "sensible" to backtracking. As an example, let us consider the following program fragment, where we assume that `x` and `y` are uninitialized logical variables, and `r`, `s1`, `s2` are sets.

```
Solver.solve(r.inters(s1,s2));
if (!r.empty()) Solver.add(y.ge(1));
else Solver.add(y.eq(0));
...
Solver.add(x.eq(0));
Solver.solve();
```

If `r` is uninitialized and `s1`, `s2` are initialized to $\{1\}$ and $\{\mathtt{x}\}$, respectively, the first constraint `r.inters(s1,s2)` admits two distinct solutions, namely, $\mathtt{x} = 1 \wedge \mathtt{r} = \{1\}$ and $\mathtt{x} \neq 1 \wedge \mathtt{r} = \{\}$. Assuming the first solution is computed first, the **if** statement adds to the constraint store the constraint $\mathtt{y} \geq 1$. Therefore, the invocation of the solver in the last statement detects a failure. The second solution for the constraint `inters` is then taken into account, but the condition of the **if** statement is no longer evaluated. The constraint solver will examine the constraint store again, with the new value for `r`, but still with the constraint $\mathtt{y} \geq 1$ stored in it. Hence, it fails again.

Basically the problem is caused by the fact that nondeterminism in JSetL is confined to constraint solving: backtracking allows the computation to go back to the nearest open choice point *within* the constraint solver, but it does not allow to re-execute user program code (the **if** statement in the above example).

The problem can be solved using the class NewConstraints. The piece of code that requires to be re-executed is defined as the body of a new constraint in the class NewConstraints, with proper parameter passing:

```
public static void testSize(Set s, Lvar n) {
    if (!s.empty()) Solver.add(n.ge(1));
    else Solver.add(n.eq(0));
    return;
}
```

Then the sample program fragment is modified as follows:

```
Solver.solve(r.inters(s1,s2));
Solver.solve(NewConstraints.testSize(r,y));
...
Solver.add(x.eq(0));
Solver.solve();
```

With these changes, the constraint testSize is evaluated again when backtracking takes place, and the **else** alternative is executed. Thus the computation of the program fragment terminates with success.

Observe that the definition of the method testSize shown above is not the actual Java code which is executed. The real code is obtained through a straightforward program transformation of the class NewConstraints. This transformation adds the auxiliary definitions needed to make the new constraint methods known to the solver and to allow the solver to pass data to them. The actual code required by the definition of the new constraint testSize is shown in Appendix A.

Another, more sophisticated, usage of the new constraint facility in JSetL is connected with the possibility for the user to implement his/her own nondeterministic procedures. This is obtained by defining the required procedures as new (nondeterministic) constraints, that is, as methods of the class NewConstraints that exploit the JSetL facilities to express and handle nondeterminism. We assume a special construct is available to express a nondeterministic choice among two or more statements:

$$\textbf{\textit{either}}\ \mathtt{S}_1\ \textbf{\textit{orelse}}\ \mathtt{S}_2 \ldots \textbf{\textit{orelse}}\ \mathtt{S}_\mathtt{n},$$

where $\mathtt{S}_1,\ldots,\mathtt{S}_\mathtt{n}$ are Java statements. The logical meaning of this construct is the disjunction $\mathtt{S}_1\vee,\ldots,\vee\ \mathtt{S}_\mathtt{n}$, while its computational interpretation is that of exploring, through backtracking, all possible alternatives $\mathtt{S}_1,\ldots,\mathtt{S}_\mathtt{n}$, starting from $\mathtt{S}_1$. The *either-orelse* construct can be used only within the class NewConstraints. This construct must be replaced, through a straightforward program transformation, by usual Java code endowed with JSetL facilities for handling nondeterminism.[8]

To illustrate these facilities we make use of a simple example: a fully nondeterministic version of the list concatenation procedure concat shown in Section 3.4. The actual code for the constraint concat is shown in Appendix A.

Example 7.1 *Nondeterministic list concatenation. List* L3 *is the concatenation of lists* L1 *and* L2*.*

[8] The program transformation of the class NewConstraints could be easily automated through the use of a suitable preprocessing tool which automatically generates the actual code of the class NewConstraints. At present, this tool is not provided by the JSetL library; hence, the required program transformation must be performed at hand.

```
class NewConstraints {
    public static void concat(Lst l1, Lst l2, Lst l3) {
        either {
            Solver.add(l1.eq(Lst.empty));
            Solver.add(l2.eq(l3));
        }
        orelse {
            Lvar x    = new Lvar();
            Lst l1new = new Lst();
            Lst l3new = new Lst();
            Solver.add(l1.eq(l1new.ins1(x)));
                                // l1 = [x | l1new]
            Solver.add(l3.eq(l3new.ins1(x)));
                                // l3 = [x | l3new]
            Solver.add(concat(l1new,l2,l3new));
        }
        return;
    }
}
```

This new version of the method `concat` *can be used not only to compute* `l3` *out of* `l1` *and* `l2` *or to test whether* `l3` *is the concatenation of* `l1` *and* `l2`*, but also to compute (nondeterministically) all possible lists* `l1` *and* `l2` *from* `l3`*. Such flexibility is obtained by using unification, instead of standard assignment, and nondeterminism, instead of a deterministic* **if** *statement.* □

The following code shows the definition of a method, called `prefix`, which uses `concat` to split a list into two sub-lists.

Example 7.2 *Check whether a list* `l1` *is a prefix of a list* `l2`*.*

```
public static void prefix(Lst l1, Lst l2)
throws Failure {
    Lst l = new Lst();
    Solver.add(NewConstraints.concat(l1,l,l2));
    Solver.solve();
    return;
}
```
□

The ability to express nondeterminism allows the user to define methods which are completely "invertible", that is, that do not impose any distinction between input and output parameters. Invertibility contributes not only to enhance reusability of the same methods, but also it provides a valuable support to declarativeness. As a matter of fact, an "invertible" method, such as the method `concat` of Example 7.1, admits a complete declarative reading, and as such it can be used in a program with no concern about the instantiation of its parameters.

Furthermore, nondeterminism can be conveniently exploited whenever one has to implement an algorithm which requires to explore a search space. Again, the presence of a high-level control abstraction such as that represented by the ***either-orelse*** construct can be of great help to devise solutions in a highly declarative style. As an example, the following is a completely nondeterministic method to find a path between two nodes in a directed labeled graph. To allow nondeterminism to be fully exploited, `path` is defined as a new constraint in the class `NewConstraints`.

Example 7.3 *Finding a path in a graph*

Given a directed labeled graph G, the problem consists in determining whether there is a path in G starting from a source node `s` *and ending in a destination node* `d`. *A directed labeled graph G can be represented as a set of pairs $\langle n_1, n_2\rangle$, where n_1, n_2 are nodes and each pair represents an edge of the graph G. This representation has an immediate implementation using JSetL's data structures: G can be implemented as a set whose elements are lists of two nodes (e.g., single characters or strings). For example, if G contains the list $['a','b']$ it means that the graph contains an arc from node $'a'$ to node $'b'$ (note that arcs can be conveniently implemented using sets—instead of lists—if G is an undirected graph).*

```
public static void path(Set G, Lvar s, Lvar d) {
    either {
        Lst finalEdge = Lst.empty.ins1(s).insn(d));
        Solver.add(finalEdge.in(G));         // [s,d] ∈ G
    }
    orelse {
        Lvar x = new Lvar();
        Lst intermediateEdge = Lst.empty.ins1(s).insn(x));
        Solver.add(intermediateEdge.in(G)); // [s,x] ∈ G, x uninit'ed l. var.
        Solver.add(path(G,x,d);             // there is a path between x and d
    }
    return;
}
```

As an example, if `g1` *is the graph*

`{['a','b'],['a','c'],['c','b'],['b','d'],['c','d']}`,

`s` *is an* `Lvar` *with value* `'a'`, *and* `d` *is an uninitialized variable, the invocation*

`Solver.solve(NewConstraints.path(g1,s,d))`

nondeterministically assigns to `d` *the values* `'b'`, `'c'`, `'d'`, *which represent all nodes reachable from* `'a'`. □

8 Related work

Our work is mainly related to proposals that define OO libraries to support, at some extent, general purpose declarative programming. In particular, various similarities exist between our proposal and proposals for libraries supporting constraint programming, such as ILOG, Koalog, Choco, JCL, JSolver. As already noticed, however, most of these proposals are mainly interested in efficient constraint solving. No one of them provide the user with a comprehensive collection of facilities to support general purpose declarative programming. In particular JSolver has many similarities with our proposal as concerns constraint handling. With respect to JSetL, however, JSolver lacks: partially specified dynamic data structures, such as lists and sets; the `forall` method; the `setof` method; the possibility to define new constraints, possibly using nondeterminism.

The library-based approach has been adopted also in other proposals to support different forms of declarative programming in the context of conventional OO programming languages, such as, functional reactive programming (e.g., Frappè [6]), definitional programming (e.g., Gisela [15]), logic programming (e.g., `tu`Prolog [8]). In particular, both Gisela [15] and `tu`Prolog [8] provide the implementation of a specific declarative language as a library of an existing OO language (a definitional programming language and Prolog, respectively). Although programs written in the supported declarative language and those written in the host language may be highly integrated—through the use of the various abstraction mechanisms provided by the host language—the declarative language is completely distinct from the host one. Conversely, in our proposal we have just one language, namely Java, endowed with additional facilities to support declarative programming in Java.

Similarities also exist between our proposal and some new general purpose declarative languages, such as Alma-0, DJ, Singleton, Flow Java. In particular JSetL and Singleton [21] share many important features—like sets and set constraints—since both proposals are at some extent based on the CLP($\mathcal{SET}$) language. SINGLETON combines most of the features considered in this paper with traditional features of imperative programming languages, such as the iterative control structures and the block structure of programs. SINGLETON, however, is a completely new language, with its own syntax and its own semantics. As already noticed, the design and development of a new language is, in general, much more demanding than the development of a library for an existing language. A comparison of pros and cons of the library-based approach with respect to the new language-based approach as concerns definition and usability of the new facilities constitutes a challenging problem which, however, is left for future work.

9 Conclusions and future work

We have presented the main features of the JSetL library and we have shown how they can be used to write programs that exhibit a quite good declarative reading, while maintaining all the features of conventional Java programs. In particular we have described the (set) constraint handling facilities supported by our library and we have shown how constraint solving can be accomplished, and how it interacts with the usual notion of program computation. Furthermore we have shown how to exploit nondeterminism, possibly by introducing new user-defined constraints. JSetL is fully implemented in Java and is available at URL `www.math.unipr.it/~gianfr/JSetL`.

Facilities provided by JSetL coexist with conventional and object-oriented programming constructs of Java. Actually, the programmer can exploit the abstraction and program structuring mechanisms of Java to separate declarative code from the more conventional one. A Java program using JSetL is likely to be composed of classes whose methods are defined in a declarative way, along with classes whose methods contain (only) conventional imperative code. Both kinds of methods can be used and understood as usual Java methods. The declarative ones, however, admit also a more abstract, declarative reading, which is essentially based on the declarative computational model of the CLP($\mathcal{SET}$) language.

Although separating the declarative methods from the imperative ones should provide better understanding of the program, constraints and other JSetL facilities for declarative programming can be freely mixed with the usual Java constructs. This opens a number of interesting problems, such as the mutual interaction between logical and conventional variables, the use of the assignment statement over logical variables, the interaction between conventional constructs and the backtracking mechanism used to implement nondeterminism, and so on. We have addressed only partially some of these problems in this paper. Very few attention is devoted to them in the literature (see for instance [2]). A deeper analysis of these problems is left for future work.

As noted in Section 1 , computation efficiency has not been a primary requirement in the development of JSetL. Actually, various improvements to the current implementation can be devised, such as the use of more sophisticated techniques to handle choice points and backtracking, the use of a more efficient data representation and processing of sets and lists when all elements are completely specified, and so on. A more critical source of possible inefficiency, however, lies in the kind of constraint solving technique supported at present by our constraint solver, which is basically a "generate & test" technique. To overcome limitations of this approach we plan to integrate our (set) constraint solver with a more efficient constraint solver over *finite domains* (FD), as suggested in [7].

On a different side, another concrete improvement could be obtained by using flexible preprocessing tools for the Java language. This would allow us to develop suitable syntax extensions that would make it simpler and more natural using the JSetL facilities.

Finally, making the solver an instantiable class (not a static one, as in the first release) opens a number of interesting new possibilities, that could be explored as future work, such as: more solvers can coexist and cooperate; solvers can be executed as different threads; constructors of the solver class can be parametric with respect to constraint solving policies.

10 Acknowledgments

The work is partially supported by MIUR project *AIDA* and by GNCS 2005 project on constraints and their applications.

References

[1] S. Abdennadher, E. Krämer, M. Saft, and M. Schmauss. Jack: A Java constraint kit. In *WFLP 2001—International Workshop on Functional and (Constraint) Logic Programming*, 2001.

[2] K.R. Apt, J. Brunekreef, V. Partington, and A. Schaerf. Alma-0: An imperative language that supports declarative programming. *ACM TOPLAS*, 20(5), 1014–1066, 1998.

[3] M. Cadoli, G. Ianni, L. Palopoli, A. Schaerf, and D. Vasile. NP-SPEC: An Executable Specification Language for Solving all Problems in NP. *Computer Languages*, Vol. 26, Pergamon, Elsevier, 165-195, 2000.

[4] Choco. Available at `http://www.choco-constraints.net`.

[5] A. Chun. Constraint programming in Java with JSolver. In *Proc. Practical Applications of Constraint Logic Programming, PACLP99*, 1999.

[6] A. Courtney. Frappé: Functional reactive programming in Java. In *Practical Aspects of Declarative Languages, PADL 2001*, LNCS, Vol. 1990, Springer-Verlag, 29–44, 2001.

[7] A. Dal Palù, A. Dovier, E. Pontelli, and G. Rossi. Integrating Finite Domain Constraints and CLP with Sets. In *PPDP'03 — Proc. of the Fifth ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, ACM Press, 219–229, 2003.

[8] E. Denti, A. Omicini, and A. Ricci. `tu`Prolog: a light-weight Prolog for internet applications and infrastructures. In *Practical Aspects of Declarative Languages, PADL 2001*, LNCS, Vol. 1990, Springer-Verlag, 184–198, 2001.

[9] M. Dincbas, P. Van Hentenryck, H. Simonis, and A. Aggoun. The constraint logic programming CHIP. In *Proc. of the 2nd Int'l Conf. On Fifth Generation Computer Systems*, 683-702, 1988.

[10] A. Dovier, C. Piazza, E. Pontelli, and G. Rossi. Sets and constraint logic programming. *ACM TOPLAS*, 22(5), 861–931, 2000.

[11] A. Dovier, C. Piazza, and G. Rossi. A uniform approach to constraint-solving for lists, multisets, compact lists, and sets. Research Report "Quaderno del Dipartimento di Matematica", n.235, Università di Parma, September 2000. (revised version available at `http://arxiv.org/abs/cs.PL/0309045`).

[12] A. Dovier, E. Pontelli, and G. Rossi. Set unification. To appear in *Theory and Practice of Logic Programming* (available at `http://arXiv.org/abs/cs/0110023`).

[13] F. Drejhammar, C. Schulte, P. Brand, and S. Haridi. Flow Java: Declarative Concurrency for Java In: C. Palamidessi, ed., *Proc. of the 9th Int. Conf. on Logic Programming*, Lecture Notes in Computer Science, vol. 2916, Springer-Verlag, 346-360, 2003.

[14] ECLiPSe, User Manual. Tech. Rept., Imperial College, London. August 1999. Available at `http://www.icparc.ic.ac.uk/eclipse`.

[15] G.Falkman, O.Torgersson. Enhancing Usefulness of Declarative Programming Frameworks through Complete Integration. In *Proc. of the 12th Int. Workshop on Logic Programming Environments*, July 2002 (available at `http://xxx.lanl.gov/abs/cs.SE/0207054`).

[16] ILOG Optimisation Suite - White Paper. Available at `http://www.ilog.com/products/optimisation/tech/optimisation/whitepaper.pdf`.

[17] JCL - the Java Constraints Library. Available at `http://liawww.epfl.ch/JCL/`.

[18] Koalog. Available at `http://www.koalog.com/php/index.php`.

[19] E.Panegai, E.Poleo, and G.Rossi. JSetL User's Manual - Version 1.0. Research Report "Quaderno del Dipartimento di Matematica", n.384, Università di Parma, November 2004.

[20] J.-F. Puget, M. Leconte. Beyond the Glass Box: Constraints as Objects. In *Proc. of the 1995 Int'l Symposium on Logic Programming*, MIT press, pp. 513-527.

[21] G. Rossi. Set-based Nondeterministic Declarative Programming in SINGLETON. In *11th Int.l Workshop on Functional and (constraint) Logic Programming*, Electronic Notes in Theoretical Computer Science, Vol. 76, Elsevier Science B. V., 17 pages, 2002.

[22] G. Smolka. The Oz programming model. In *Current Trends in Computer Science*, J. van Leeuwen, Ed., LNCS, vol. 1000, Springer-Verlag, 1995.

[23] P. Van Roy, S. Haridi. *Concepts, Techniques, and Models of Computer Programming.* The MIT Press, 2004.

[24] Neng-Fa Zhou. DJ: Declarative Java, Version 0.5, User's manual. Kyushu Institute of Tecnology, 1999. Available at `http://www.cad.mse.kyutech.ac.jp/people/zhou/dj.htlm`.

[25] Neng-Fa Zhou. Building Java Applets by using DJ—a Java Based Constraint Language. Available at `http://www.sci.brooklyn.cuny.edu/~zhou`.

A Implementing user-defined constraints

In this section we present the actual Java code used to implement the user-defined constraints `testSize` and `concat`.

```
class NewConstraints
   {
   SolverClass Solver;
   public NewConstraints(SolverClass CurrentSolver) {
       Solver = CurrentSolver;
       }

   public static Constraint testSize(Set s, Lvar n) {
       return new Constraint("testSize",x);
       }
   public static Constraint concat(Lst l1, Lst l2, Lst l3) {
       return new Constraint("concat",l1,l2,l3);
       }

   protected void user_code(Constraint c)
   throws Failure, NotDefinedConstraint {
       if(c.getName() == "testSize") testSize(c);
       else if(c.getName() == "concat") concat(c);
       else throw new NotDefinedConstraint();
       return;
       }
```

```
protected void testSize(Constraint c)
throws Failure {
    Set s = (Set)c.getArg(1);
    Lvar n = (Lvar)c.getArg(2);
    if (!s.empty()) Solver.add(n.ge(1));
    else Solver.add(n.eq(0));
    return;
    }

protected void concat(Constraint c)
throws Failure {
    Lvar l1 = (Lvar)c.getArg(1);
    Lvar l2 = (Lvar)c.getArg(2);
    Lvar l3 = (Lvar)c.getArg(3);
    switch(c.getAlternative())
        {
        case 0:
            Solver.addChoicePoint(s);
            Solver.add(l1.eq(Lst.empty));
            Solver.add(l2.eq(l3));
            return;
        case 1:
            Lvar x = new Lvar();
            Lst l1new = new Lst();
            Lst l3new = new Lst();
            Solver.add(l1.eq(l1new.ins1(x)));
                                        // l1 = [x | l1new]
            Solver.add(l3.eq(l3new.ins1(x)));
                                        // l3 = [x | l3new]
            Solver.add(concat(l1new,l2,l3new));
            return;
        }
    }
}
```

The variable `Solver` is used to refer to the current solver. Its value is specified whenever an instance of the class `NewConstraints` is created within the current solver, and refers to the solver itself. The two static methods `testSize` and `concat` create two 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`, whereas the method `add` takes an object of this class as its parameter. Each different type of constraint is identified by a string name (e.g., `"concat"`), which can be specified as a parameter of the constraint constructor and can be read through the method `getName` of the class `Constraint`.[9]

The method `user_code` acts as a "router": it associates each constraint name with the corresponding method that implements the user-defined constraint. These methods must, first of all, retrieve the constraint arguments, whose number and

[9]At present, the only public method of the class `Constraint`, besides the constructors, is the method `and` that it is used to build a conjunction of two constraints (actually, it returns an object of class`ConstraintsConjunction`). All other members are protected and therefore can be used only within the JSetL package (in particular, within the class `NewConstraints`).

type depend on the constraint itself. Moreover, if the constraint is nondeterministic, the different alternatives occurring in the *either-orelse* construct are implemented as different cases of a `switch` statement, using the control information stored in the constraint itself (and obtainable through the method `getAlternative` of the class `Constraint`). Each `case` block, but the last one, creates a choice point and adds it to the stack of the alternatives by invoking the method `addChoicePoint` of the class `SolverClass`. Then the remaining code of the `case` block implements the constraint rewritings as specified in the abstract definition of the new constraint.