Available online at www.sciencedirect.com
SCIENCE DIRECT®
Science of Computer Programming 57 (2005) 217–250
Science of Computer Programming
www.elsevier.com/locate/scico

Multi-paradigm Java–Prolog integration in tuProlog

Enrico Denti[a,*], Andrea Omicini[b], Alessandro Ricci[b]

[a] *DEIS, Dipartimento di Elettronica, Informatica e Sistemistica, Alma Mater Studiorum, Università di Bologna, Viale Risorgimento 2, 40136 Bologna, Italy*
[b] *DEIS, Dipartimento di Elettronica, Informatica e Sistemistica, Alma Mater Studiorum, Università di Bologna, Via Venezia 52, 47023 Cesena, Italy*

Received 20 February 2004; received in revised form 5 November 2004; accepted 1 February 2005
Available online 11 March 2005

Abstract

tuProlog is a Java-based Prolog engine explicitly designed to be minimal, dynamically configurable, and support full and clean Prolog/Java integration. In this paper, we discuss the tuProlog approach to Prolog/Java multi-paradigm integration. After tuProlog motivations and requirements, we present some examples of bidirectional Prolog/Java integration and discuss the model and architecture of the tuProlog system. Then, we focus on the specific issue of the access to Java resources from tuProlog, discuss the essentials of its implementation, and compare it extensively with many other relevant related approaches and systems.

Keywords: Java; Prolog; Language integration; Reflection; Agent infrastructures

1. Motivations and requirements

tuProlog [6,26] is a Java-based framework providing support to logic-based technology in the form of a Prolog virtual machine (VM henceforth) built on top of the standard Java VM. Meant to work as the core technology for both Internet application components and infrastructures, tuProlog is designed to feature the following key properties:

* Corresponding author. Tel.: +39 051 2093015; fax: +39 051 2093073.
E-mail addresses: enrico.denti@unibo.it (E. Denti), andrea.omicini@unibo.it (A. Omicini), a.ricci@unibo.it (A. Ricci).

0167-6423/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2005.02.001

- *Minimality*. tuProlog is required to be as thin and light-weight as possible: to this end, it is designed as a pure inferential engine, available as a Java class through a simple interface. This feature is particularly relevant for use in small devices such as PDAs or mobile phones, and as a support technology for Internet-based infrastructures.
- *Dynamic configurability*. tuProlog's choice of minimality calls for a high degree of configurability as its necessary counterpart. In particular, configurability should be *dynamic*, so as to face the openness of most application environments such as the Internet, and enable both static and dynamic configuration of components in a uniform way. It should also be *open*, in the sense that nothing – no complex mechanisms, or static declarations – should prevent or just make it difficult to define and enact new add-ons.
- *Full & clean Prolog/Java integration*. While providing some form of integration between Prolog and Java is today a must for practically any mainstream Prolog engine [16], defining a full, bidirectional, easy-to-use integration scheme is a rather more complex task. tuProlog's aim is to provide access to Java resources (objects, classes and packages) in a dynamic way, with no setup intricacies or static constraints. Dually, tuProlog engines are meant to be exploited straightforwardly from the Java code, yet with no setup intricacies or static pre-declarations. At the same time, a key design requirement is that such integration does not mix the logic and the object-oriented paradigms, nor should it alter in any way the very nature of either language – neither Prolog, nor Java. This constraint is introduced not only for conceptual cleanness, but because we believe that only a simple, non-intrusive integration scheme can actually be used in an effective way, preserving and promoting the power of both paradigms and technologies.

The accomplishment of the above requirements makes tuProlog an effective enabling technology for the development of Internet components, applications, and infrastructures [6]: indeed, it is the base building block for the TuCSoN [17] and LuCe [5] coordination infrastructures.

In order to make tuProlog feature such properties, some non-trivial design and development problems need to be solved. For instance, how can we provide run-time linking and discharging of previously-unknown Java code representing well-formed tuProlog libraries? Which patterns allow such a flexible run-time configurability, while keeping the (uncoupled) development of linked libraries simple and natural? How can we support the run-time creation and mutual interaction of Java objects (instances), without knowing a priori the involved Java classes, and without affecting the minimality and elegance of the core?

In this article, we report on how these requirements were successfully met in the tuProlog project, by carefully designing a minimal Prolog core, along with a clean and expressive extension mechanism which makes it possible to load/unload libraries dynamically, including support for dynamic compilation. The core of this article is the in-depth discussion of how tuProlog fruitfully combines Prolog and Java, by enabling a seamless integration of the logic and object-oriented imperative paradigms, while preserving orthogonality in the paradigm integration. Integration is promoted at two different levels:

- at the system level, both Java and Prolog can be combined to build tuProlog libraries, which can therefore be designed taking the best of the two worlds;
- at the application level, bidirectional Prolog/Java integration in tuProlog enables both Java items to be accessed effectively and simply from a Prolog program, without static pre-compilations or pre-declarations, and, conversely, Prolog engines to be dynamically instantiated, configured and exploited from a Java program.

An intriguing aspect of the Java/Prolog synergy is that it enables and promotes new expressive forms, that would not be possible in the Java world alone: as one example, Prolog can easily provide the ability to control failure-driven loops via backtracking, where the iterated operation could be performed by Java, thus de facto enriching Java programs with non-determinism – *yet* maintaining the two paradigms clearly separated, since neither language is 'polluted' by alien concepts from the other language paradigm. Moreover, since Prolog is interpreted, while Java is compiled, a Prolog-driven Java program is somehow close to a 'semi-interpreted' Java program, where Prolog can drive in an interpreted way the computational flow of compiled Java actions.

2. Integration examples

Let us start by having a look at three simple examples, one for each of the three main forms of Java/Prolog integration supported by tuProlog: *Prolog from Java*, *Java for Prolog*, and *Java from Prolog*. *Prolog from Java* means the possibility of exploiting Prolog engines as simple Java objects. *Java for Prolog* is concerned with using Java as the implementation language for writing new tuProlog libraries. *Java from Prolog* refers to the ability of manipulating Java objects from tuProlog programs in a straightforward way. Altogether, they constitute what we mean as 'full and clean Java/Prolog integration', which is tuProlog's key feature with respect to other Java-based Prolog implementations (comparisons are widely discussed in Section 6).

2.1. Prolog from Java

Exploiting Prolog engines as simple and self-contained Java objects makes it possible, for instance, to embed high-level symbolic reasoning abilities in a Java software component, as well as to bring non-determinism to the Java world in a non-intrusive, controlled way, while maintaining the separation between the logic and the object-oriented paradigms. For this purpose, it should be possible to create multiple independent engines, each configured with its own set of theories and libraries. This leads to the following requirements:

(1) Prolog engines should expose a minimal interface – i.e., just what is needed to load a theory, load and unload libraries, and demonstrate goals;
(2) any static constraint or pre-declaration should be carefully avoided;
(3) an intuitive and simple mapping of the basic Prolog data types (terms, structures, numbers, variables, ...) onto Java classes should be provided.

As an example of the desired immediacy, let us refer to Table 1, where a Java application exploits a Prolog engine to perform a symbolic manipulation (computing the first derivative

Table 1
Configuration and exploitation of a Prolog engine in Java *(top)*, and the Prolog theory `math.pl` for symbolic derivation and evaluation of math expressions *(bottom)*

```
1   Prolog engine = new Prolog();
2   Theory t = new Theory(new java.io.FileInputStream("math.pl"));
3   engine.setTheory(t);
4   SolveInfo answer = engine.solve("dExpr(sin(2*x)*cos(x), Der)");
5   Term derivative = answer.getTerm("Der");
6   Term newGoal = new Struct("evalExpr", derivative,
                              new Double(0.5), new Var("X"));
7   SolveInfo result = engine.solve(newGoal);
8   double value = ((Number)result.getTerm("X")).getDouble();
```

```
% math.pl
dExpr(T,DT)                       :- dTerm(T,DT).
dExpr(E+T,[DE+DT])                :- dExpr(E,DE), dTerm(T,DT).
dExpr(E-T,[DE-DT])                :- dExpr(E,DE), dTerm(T,DT).
dTerm(F,DF)                       :- dFactor(F,DF).
dTerm(T*F,[[DT*F]+[T*DF]])        :- dTerm(T,DT), dFactor(F,DF).
dTerm(T/F,[[F*DT]-[T*DF]]/[F*F]) :- dTerm(T,DT), dFactor(F,DF).
dFactor(x,1).
dFactor(N,0)                      :- number(N).
dFactor([E],DE)                   :- dExpr(E,DE).
dFactor(-E,-DE)                   :- dExpr(E,DE).
dFactor(sin(E), [cos(E)*DE] )     :- dExpr(E,DE).
dFactor(cos(E), [-sin(E)*DE] )    :- dExpr(E,DE).

evalExpr(T,V,R)         :- evalTerm(T,V,R).
evalExpr(E+T,V,R)       :- evalExpr(E,V,R1), evalTerm(T,V,R2),
                           R is R1+R2.
evalExpr(E-T,V,R)       :- evalExpr(E,V,R1), evalTerm(T,V,R2),
                           R is R1-R2.
evalTerm(F,V,R)         :- evalFactor(F,V,R).
evalTerm(T*F,V,R)       :- evalTerm(T,V,R1), evalFactor(F,V,R2),
                           R is R1*R2.
evalTerm(T/F,V,R)       :- evalTerm(T,V,R1), evalFactor(F,V,R2),
                           R is R1/R2.
evalFactor(x,V,V).
evalFactor(N,V,N)       :- number(N).
evalFactor([E],V,R)     :- evalExpr(E,V,R).
evalFactor(-E,V,R)      :- evalExpr(E,V,R1), R is -R1.
evalFactor(sin(E),V,R) :- evalExpr(E,V,R1), R is sin(R1).
evalFactor(cos(E),V,R) :- evalExpr(E,V,R1), R is cos(R1).
```

of a function) which is much easier in Prolog than in Java. At the same time, the Java-based nature of the application makes it possible to use the standard Java classes for the other tasks, such as loading the Prolog theory from a file, possibly adding a GUI, etc. So, a Prolog engine is created and configured by loading the proper Prolog theory `math.pl` (lines 1–3), which defines predicates for symbolic derivation and expression

evaluation.[1] Then, the derivative of the desired expression – possibly read from a GUI, although here we assume the expression `sin(2*x)*cos(x)` – is computed (lines 4–5) and evaluated for `x=0.5` (lines 6–7), leaving the result in a standard Java variable (line 8). With respect to the above requirements:

- The minimality of the tuProlog interface makes it possible to create and exploit a Prolog engine by means of just the default constructor and two simple methods, `setTheory` and `solve`; in turn, the minimality of `Theory` methods makes the creation of a new theory from a text file straightforward.
- Throughout the application code, no static declarations exist in either Java or the Prolog part: the whole application is just the 8-line Java source plus the Prolog theory. Moreover, as an external text file, the Prolog theory can be changed without touching the Java side at all.
- Prolog types are mapped onto suitable tuProlog Java classes: namely, Prolog terms correspond to `Term` objects, Prolog structures to `Struct` objects, Prolog variables to `Var` objects, and Prolog numbers to `Number` objects.

2.2. *Java for Prolog*

tuProlog allows Java to be exploited also as an implementation language for writing new tuProlog libraries in a simple, disciplined way. By doing so:

- the intrinsic modularity of the Java class concept can be exploited to group conceptually-related built-ins into the same library;
- the Java language features can be used to keep the Java/Prolog mapping as simple and intuitive as possible, minimising the burden on developers in terms of both classes to define and amount of code to write;
- valuable documentation, based on Java comments, can be automatically produced via JavaDoc – a not-so-obvious issue in approaches that do not map a set of built-ins into a Java class.

Following the conventions detailed in Section 4, Table 2 defines a new library (`StringLibrary`) implementing the new Prolog predicate `to_lower_case/2`. To use the new predicate, the engine just needs the proper `.class` file in its JVM's classpath *at run time* – no static information. So, the available predicates of an engine can be enriched *dynamically* by just adding new `.class` files to its JVM's classpath – a feature which is particularly interesting when combined with tuProlog support for dynamic compilation.

Libraries can be loaded and unloaded dynamically: once loaded, the new predicates remain available until the library is explicitly unloaded, and can be invoked just like any other predicate. For instance, loading `StringLibrary` and invoking the new predicate `to_lower_case/2` would look like[2]:

[1] Square brackets are used instead of round parentheses to avoid clashes with the Prolog parser.

[2] By default, the library name is equal to the class name; however, they may be different if the Java implementation of the library specifies another name. This feature is useful to manage library versions, as shown in Section 3.2.

Table 2
Writing a new tuProlog library in Java

```
public class StringLibrary extends Library {
    public boolean to_lower_case_2(Term source, Term dest){
        String st = source.toString().toLowerCase();
        return unify(dest, new Struct(st));
    }
}
```

```
?- load_library('StringLibrary', LibName).
yes. LibName / 'StringLibrary'
?- to_lower_case('ABC', LowercaseString).
yes. LowercaseString / abc
```

As a result, the (Prolog-based) language can be dynamically extended, accessing any Java resource without requiring any special knowledge about the Java world: the two paradigms are "mixed" and led to cooperate, but in such a (controlled) way that each language still operates as usual on its own side.

2.3. Java from Prolog

The third form of integration is represented by the possibility of creating and interacting with Java objects from Prolog programs. It should be clear that our goal is *not* to model the object-oriented paradigm in the logic framework *in general* – which is a complex issue, widely investigated in the literature – but just to be able to create and use Java objects from the tuProlog language in a simple and dynamic way. More precisely, the form of Java/Prolog integration we devise is intended to support both object construction and interaction (method call) from a declarative context, while keeping the two worlds/paradigms clearly separate. This means that we do not model the notions of class and object as tuProlog abstraction: instead, we define some logic constructs (namely, operators) whose side effect – *outside* the Prolog world – is the creation of, or the interaction with, a Java object.

In this context, preserving the intrinsic dynamism of the Java world means maintaining *also for Prolog* the Java feature that classes need be available only at runtime, when requested, with no static pre-declarations or pre-compilations of any kind. Also, preserving the agility of the Java language means that both object creation and method invocation should be expressed and occur in tuProlog as similarly as possible to a standard Java program, even if tuProlog does not provide directly the notions of object and object reference.

So, in Table 3 the `java_object/3` predicate performs instance creation, while the special `<-/2` and `(<-,returns)/3` operators invoke methods on the object instance represented by the Prolog term on their left – also retrieving the method result in the latter case. As a result, a three-line Prolog program is all that is needed to open a Swing dialog for file selection (a `javax.swing.JFileChooser` instance), show it on screen (line 3), and retrieve the user-selected file (line 4), yet keeping a form of separation between the two worlds. In tuProlog, Java objects only exist on the Java side, while the Prolog side

Table 3
Creating and exploiting a Java Swing object from tuProlog

```
1    choose_file(File) :-
2        java_object('javax.swing.JFileChooser', [], Dialog),
3        Dialog <- showOpenDialog(_),
4        Dialog <- getSelectedFile returns File.
```

Table 4
The visitor pattern in a multi-paradigm Java/Prolog approach

```
1    applyToAll(Iterator,Method):-
2       repeat, applyToOne(Iterator,Method).
3    applyToOne(Iterator,_):-
4       Iterator <- hasNext returns false,!.
5    applyToOne(Iterator,Method):-
6       Iterator <- next returns Obj, Obj <- Method, fail.
```

handles just standard Prolog atoms: the two worlds come to contact only in well-known, controlled points, like the `java_object/3`, `<-/2` and (`<-`, `returns`)`/3` predicates shown in the example. There, and only there, the Prolog atoms are interpreted as references to underlying Java objects, and cause side effects in the Java world. So, the Prolog variable `Dialog` is bound by `java_object/3` to a system-generated Prolog atom, whose mapping to the underlying `JFileChooser` instance comes to surface only when invoking methods (lines 3–4). The Prolog variable `File` is bound to another system-generated Prolog atom, whose mapping to the `java.io.File` object returned by `getSelectedFile` eventually comes to surface only when the caller of `choose_file/1` refers to it inside one of tuProlog's special predicates.

An extra value of the synergy between Prolog and Java is that new expressive forms are now available that would not be possible in the Java world alone: in particular, although a Java function is inherently deterministic, a Java program can be enriched de facto with non-determinism by just having it call a non-deterministic Prolog predicate. Table 4 shows this aspect by realising the visitor pattern: there, Prolog provides the ability to control failure-driven loops via backtracking, tuProlog adds the chance to represent methods as first-class objects and the integration with Java, and Java applies the operation. So, Java programs can borrow non-determinism from the Prolog virtual machine, yet without mixing the logic and the object-oriented paradigms: in fact, we explicitly excluded that non-deterministic predicates could be implemented directly in Java, since this would alter the semantics of both languages.

Moreover, a Java program that embeds a tuProlog engine features the intriguing property of being 'semi-interpreted', in the sense that its behaviour can be modified not only by changing and recompiling Java classes, as usual, but also by suitably changing the Prolog theory – an external text file. As a result, provided that the hybrid application is suitably engineered, the application behaviour can be changed without touching the Java compiled classes. tuProlog support for dynamic compilation will provide one step further towards dynamism and 'interpreted-like' behaviour.

3. The tuProlog system

In this Section, we first focus on the tuProlog core, discuss its structure with respect to the issue of making Prolog engines exploitable as Java objects, and present the corresponding Java interfaces. Then, we focus on dynamic configurability and discuss how Java can be exploited as the implementation language for new libraries, presenting tuProlog standard libraries (Section 3.2), and outlining how new libraries can be defined and used (Section 3.3). We illustrate in particular how new libraries can be developed exploiting either a Java-only, or a hybrid Java + Prolog approach, and present the mapping between Prolog entities and Java classes. Instead, the more complex issue of enabling Prolog programs to dynamically create Java objects and interact with them in a simple yet complete way will be addressed in detail in Section 4.

3.1. The Prolog core

The tuProlog core is a minimal inferential engine, providing just the basic inference and unification mechanisms, and the basic Prolog operators: goal conjunction, cut, arithmetics, term comparison, operator definition, clause database manipulation, predicate call – and obviously `true/0`, `fail/0`, `halt/0`. A reflection-based extension mechanism enables the core to dynamically add/remove any desired set of predicates, by loading/unloading libraries.

The engine is implemented as the Java class `Prolog` (see Table 5): a Java application can instantiate as many independent tuProlog engines as required, each configured in its own way with respect to both the clause database (theory) and the required language extensions (libraries). More precisely, the theory to be used for demonstrations can either be set from scratch by the `setTheory` method, or just added to the existing theory via the `addTheory` method: both take a `Theory` object as their argument, which can be built from an input stream, a string, or a clause list (represented as a `Struct` object). The current theory can be retrieved, in the form of a `Theory` instance, via the `getTheory` method. Analogously, libraries can be loaded and unloaded via the `loadLibrary` and `unloadLibrary` methods, whose string argument is the name of the library to be loaded or unloaded, respectively; an exception is thrown if the indicated library is invalid. A reference to one of the currently-loaded libraries can be obtained via the `getLibrary` method, whose result is a reference to (the abstract class) `Library`. Goal resolution is handled via the `solve`, `solveNext`, and `hasOpenAlternatives` methods. Both `solve` and `solveNext` take a Java object representing a Prolog term as their argument, and return a `SolveInfo` object which encapsulates information about the success or failure of the query, and, in case of success, the solution found and the corresponding variable bindings. In the case of `solveNext`, an exception is raised if no further solutions exist. An overloaded version of `solve` takes a string argument representing the text of the goal: that string is then parsed to build the corresponding `Term`, or an exception is raised if the string is malformed. Finally, the `halted` method returns `true` if the engine has stopped.

For the sake of concreteness, Table 6 shows a naïve console-based interpreter built on top of a tuProlog engine. After creation, the engine is initialised with a theory built from a text file (whose name is taken from the command line), then a classic read/solve loop is

Table 5
The main public methods of `Prolog` and `SolveInfo` classes

```
public class Prolog implements Serializable {
  ...
  public void setTheory(Theory t) throws InvalidTheoryException {...}
  public void addTheory(Theory t) throws InvalidTheoryException {...}
  public Theory getTheory() {...}
  public Library loadLibrary(String name)
                                throws InvalidLibraryException {...}
  public void unloadLibrary(String name)
                                throws InvalidLibraryException {...}
  public Library getLibrary(String name) {...}
  public SolveInfo solve(Term goal) {...}
  public SolveInfo solve(String goalAsString)
                              throws MalformedGoalException {...}
  public boolean hasOpenAlternatives() {...}
  public SolveInfo solveNext() throws NoMoreSolutionException {...}
  public boolean halted() {...}
}
```

```
public class SolveInfo implements Serializable {
  public boolean isSuccess() {...}
  public Term getTerm() throws UnknownVarException {...}
  public Term getSolution() throws NoSolutionException {...}
}
```

started. Whenever a new goal is read from the standard input, the engine's `solve` method is invoked: if multiple solutions exist, the repeated invocation of `solveNext` enables the user to explore the open alternatives. When the `halt` predicate is eventually encountered, the demonstration succeeds, and the current theory, possibly modified by user interactions, is saved to file.

3.2. Library-based configurability

The tuProlog engine is by design choice a minimal, purely-inferential core: so, there are no technically *built-in* predicates, intended as predicates statically defined inside the core. Our 'built-in' predicates are just predicates defined by some library, and are therefore 'built-in' only in the sense that, and only as long as, one of the currently-loaded libraries provides a definition for them. So, should such a library be unloaded, the corresponding 'built-in' predicates become undefined, as if they never existed. Unlike most Prolog systems, in tuProlog this also applies to the ISO predicates and evaluable functors [7]: the I/O predicates are further separated, so as to enforce orthogonality between interaction and computation. As a result, there are three standard libraries:

- `BasicLibrary`, which defines some basic predicates and functors usually found in Prolog systems, including predicates to load/unload libraries (such as `load_library`), with the only exception of I/O predicates;

Table 6
A simple console-based Prolog interpreter written in Java using tuProlog

```
import alice.tuprolog.*;
import java.io.*;

public class Console {
 public static void main (String args[]) throws Exception {
  Prolog engine=new Prolog();
  if (args.length>0)
      engine.setTheory(new Theory(new FileInputStream(args[0])));
  BufferedReader stdin =
      new BufferedReader(new InputStreamReader(System.in));
  while (true) {     // interpreter main loop
   String goal;
   do { System.out.print("?- "); goal=stdin.readLine();
   } while (goal.equals(""));
   try {
    SolveInfo info = engine.solve(goal);
    if (engine.halted()) break;
    else if (!info.isSuccess()) System.out.println("no.");
    else if (!engine.hasOpenAlternatives())
      System.out.print("yes. \n" + info + "\n");
    else { // main case
      System.out.println(info + " ?");
      String answer = stdin.readLine();
      while (answer.equals(";") && engine.hasOpenAlternatives()) {
        info = engine.solveNext();
        if (!info.isSuccess()) { System.out.println("no."); break; }
        else { System.out.println(info + " ?");
               answer = stdin.readLine();
        } // endif
      } // endwhile
      if (!answer.equals(";"))
          System.out.print("yes. \n" + info + "\n");
      else if (!engine.hasOpenAlternatives())
          System.out.println("no.");
    } // end main case
   } catch (MalformedGoalException ex) {
          System.err.println("syntax error.");
   } // end try
  } // end main loop
  if (args.length>1) {
   Theory curTh = engine.getTheory(); // save current theory to file
   new FileOutputStream(args[1]).write(curTh.toString().getBytes());
  }
 }
}
```

- `IOLibrary`, which provides the standard Prolog I/O predicates, such as `write/1`, `read/1`, and `nl/0`;
- `JavaLibrary`, which defines all the predicates for Prolog/Java integration.

Table 7
Writing a tuProlog library in Java: library name vs. class name

```
public class NewStringLibrary extends Library {
  public String getName(){ return "StringLibrary"; }
  ...
}
```

The default engine is configured so as to load them only; however, an engine can be configured in any other way, possibly with no built-ins at all, or with a different set of built-ins, tailored to a specific application.

tuProlog also supports *dynamic configurability*: after the initial configuration, an engine can be reconfigured by loading/unloading libraries on-the-fly, thus enriching/reducing the set of available 'built-ins' by need. A library can be loaded into a running engine either from Java, via the `loadLibrary` method, or from Prolog, via the `load_library/2` predicate: in both cases, the only requirement is that the proper *`class`* file is in the current JVM class path. A Java program loads a library into an engine by specifying the name of the class implementing the library: the returned reference to the loaded library can be used to unload the library later on. Analogously, a Prolog program loads a library by calling `load_library/2` with the fully-qualified name of the library class as the first argument, and a variable as the second argument: if the predicate succeeds, this variable is bound to the library name, and can be used later for unloading. So, for instance, a Java program would load into `engine` the `StringLibrary` library defined in Table 2 by invoking `engine.loadLibrary('StringLibrary')`, while a Prolog program would load it by calling `load_library('StringLibrary',Lib)`: the `Lib` variable is bound to the library name – in this case, `'StringLibrary'` again.

The choice of keeping the library name separate from the name of the class that implements the library makes it possible to define multiple versions of the same library, supporting the dynamic upgrade of a library implementation. A library can specify its name by implementing the `getName` method: by default, the library name is assumed equal to the class name. As an example, the `NewStringLibrary` class shown in Table 7 provides an alternate implementation of `StringLibrary`: to replace the current one from Prolog, one could just unload the old version and re-load the new version, like this:

```
?- unload_library('StringLibrary'), load_library('NewStringLibrary', NewLib).
   yes. NewLib / 'StringLibrary'
```

Although tuProlog libraries are expressed in Java, they are not required to be fully implemented in this language. In fact, Java-only libraries are the simplest case (Section 3.3.1), but hybrid Java + Prolog libraries are also possible, where a Prolog theory is embedded into a Java string so that the two parts cooperate to define the overall library behaviour (Section 3.3.2).

3.3. Developing libraries

This section presents the tuProlog approach to the definitions of new built-ins, along with the consequent mapping of the main Prolog concepts as Java classes; the

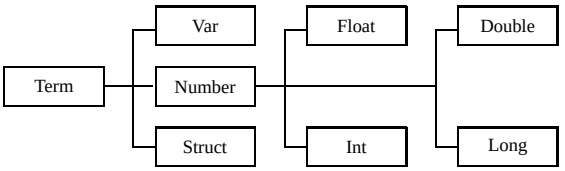

Fig. 1. tuProlog taxonomy of Prolog entities in Java.

Table 8
Some examples of how to use tuProlog Java classes mapping Prolog terms

```
    import alice.tuprolog.*;
    ...
 1  Var varX = new Var("X"), varY = new Var("Y");
 2  Struct atomP = new Struct("p");
 3  Struct list = new Struct(atomP, varY);       // should be [p|Y]
 4  System.out.println(list);              // prints the list [p|Y]
 5  Struct fact = new Struct("p", new Struct("a"), new Int(5));
 6  Struct goal = new Struct("p", varX, new Var("Z"));
 7  boolean res = goal.unify(fact);           // should be X/a, Z/5
 8  System.out.println(goal);     // prints the unified term p(a,5)
 9  System.out.println(varX);  // prints the variable binding X / a
 10 Var varW = new Var("W");
 11 res = varW.unify(varY);                        // should be Z=Y
 12 System.out.println(varY); // prints just Y, since it is unbound
 13 System.out.println(varW);  // prints the variable binding W/Y
```

opposite – enabling Prolog programs to access Java resources – is a rather more critical issue, and will be discussed in depth in Section 4.

In order to express Prolog predicates and evaluable functors in Java, we first have to define a Java mapping for the main Prolog entities: Fig. 1 shows the adopted taxonomy, which is rooted by the abstract class `Term`. This class provides common services such as term unification, term parsing, term copying, etc.; its subclasses distinguish among untyped terms (structures), numbers, and variables. `Struct` objects are characterised by a functor name (a Java string) and a list of arguments, which are `Terms` themselves and can be individually retrieved via the `getTerm` method. Prolog lists are a special case of `Struct` objects, built from either two `Terms` (the list head and tail) or an array of `Terms`; by convention, the default constructor builds the empty list. `Number` subclasses offer methods such as `intValue`, `longValue`, etc. to retrieve the number value as a suitable primitive Java value. `Var` objects are built from a Java string representing the variable name on the Prolog side: the default constructor builds the Prolog anonymous variable.

Table 8 illustrates some examples of creating and handling Prolog data from a Java program: variable creation (lines 1 and 10), list construction (lines 2–4), term construction for `p(a,5)` and `p(X,Y)` (lines 5–6), term unification (lines 7–13). It is worth noting that two different `Var` objects, even if built from the same Java name, always refer to distinct Prolog variables, except when occurring inside the same term, in which case

they unify. So, multiple occurrences of an expression such as `new Var("Y")` outside the same term do *not* refer to the same Prolog variable Y: rather, everything goes as if they were renamed Y1 and Y2. To reference the same Prolog variable in multiple places, one just has to repeatedly use the same Java identifier (e.g. `varY` in lines 1, 3, 11) instead of creating new variables. In contrast, homonymous variables inside the same term, as in `new Struct(new Var("Y"), new Var("Y"))`, as expected in Prolog, represent the same variable.

3.3.1. *Java-only libraries*

A tuProlog library exploits Java to define new *predicates* and/or *evaluable functors*. Since Java does not support non-determinism, a Java-only library is inherently deterministic: this is not to be seen as a limitation, but rather as a specific design choice to prevent undesired mix of the two paradigms. If needed, non-determinism can be achieved by building hybrid Java + Prolog libraries (Section 3.3.2), where it is confined inside the Prolog world. New libraries must extend the base abstract class `Library` and define the new built-ins according to a simple signature convention:

```
public boolean name_arity(Term a1, Term a2, ...)      (for predicates)
public Term name_arity(Term a1, Term a2, ...)         (for evaluable functors)
```

In fact, argument types can actually be any subclass of `Term`, such as `Number`; so, the actual method signature could better be written as follows:

```
public boolean name_arity(<? extends Term> a1, ...)
public Term name_arity(<? extends Term> a1, ...)
```

where the syntax inspired to the Java with wildcards [10]. Independently of the syntax used to describe it, however, the above convention is the key of tuProlog dynamic configurability, since it enables the engine, given an unknown predicate or evaluable functor, to guess the name of the Java class that implements it. In turn, this allows Java reflection to load the library class and start using it immediately, avoiding any static pre-declaration.

Table 9 (*top*) shows the definition of two new built-ins. The `sum/2` evaluable functor is implemented by method `sum_2`, whose arguments are two `Number` instances: the result is returned as a suitable `Int` instance. Analogously, method `println_1`, implementing predicate `println/1`, prints its argument and returns `true` to report success. To use these built-ins, one just loads the library bytecode, either from Prolog via `load_library/1` as in Table 9 *(middle)*, or from Java via `loadLibrary` (not shown). This approach is also compatible with inheritance: Table 9 (*bottom*) shows how to define a new library by extending `TestLibrary`, thus adding two more built-ins, `sub/2` and `even/1`.

3.3.2. *Hybrid Java + Prolog libraries*

Since Java does not inherently support non-determinism, tuProlog libraries written only in Java are necessarily deterministic. However, non-deterministic libraries – whose predicates or evaluable functors can provide multiple solutions – can also be defined in tuProlog, yet without mixing the two paradigms. The basic idea is, once again, to add a non-deterministic Prolog layer on top of the deterministic Java layer, as done in

Table 9
Declaration of `TestLibrary` *(top)*, its use in tuProlog *(middle)*, and library extension via inheritance *(bottom)*

```
import alice.tuprolog.*;
import alice.tuprolog.Number;
import alice.tuprolog.Float;

public class TestLibrary extends Library {
  // builtin functor sum(A,B)
  public Term sum_2(Number arg0, Number arg1){
    return new Int(arg0.intValue() + arg1.intValue());
  }
  // builtin predicate println(Message)
  public boolean println_1(Term arg0){
    System.out.println(arg0);
    return true;
  }
  public boolean invertCase_2(Struct input, Var output){
    String out, in = input.toString();
    if (Character.isUpperCase(in.charAt(0))){
        out = in.toLowerCase();
    } else out = in.toUpperCase();
    return output.unify(new Struct(out));
  }
  public String getTheory(){
    return "print_inverted(X) :- invertCase(X,Y), print(Y).\n";
  }
}
```

```
test :-
    load_library('TestLibrary'),
    N is sum(5,6),
    println(N),
    print_inverted('test').
```

```
public class DerivedTestLibrary extends TestLibrary {
  public Term sub_2(Number arg0, Number arg1) {
   return new Int(arg0.intValue() - arg1.intValue());
  }
  public boolean even_1(Number n) {
   return (n.intValue() % 2 == 0);
  }
}
```

Section 2 (Table 4); the difference is that now Prolog and Java are both exploited as library implementation languages, *inside* the library and *transparently* to the final library user.

To make this possible, the tuProlog base class `Library` defines the `getTheory` method, which is supposed to return (a string representation of) a Prolog theory: obviously, its default implementation in `Library` just returns the empty string. When loading a library, the tuProlog core always calls `getTheory`, too, and adds the provided theory to the engine's configuration. As a result, defining a hybrid Java + Prolog library simply amounts

Table 10
A mixed Java + Prolog library obtained via inheritance

```
public class MyLibrary extends TestLibrary {
 public String getTheory(){
  return "print(X) :- println(X).\n" +
         "print(X) :- invertCase(X,Y), print(Y).\n";
 }
}
```

at expressing in Java the deterministic, low-level code of some suitable built-ins, and redefine `getTheory` so as to return (a string representation of) a Prolog theory that defines the non-deterministic public built-ins on top of the deterministic Java built-ins. Table 10 shows a hybrid library where the theory expressed by `getTheory` defines the non-deterministic predicate `print/1` so that it provides infinite solutions, alternately printing in upper and lowercase via the deterministic `invertCase/2` and `println/1` built-ins inherited from `TestLibrary`.

4. Accessing the Java world in tuProlog

`JavaLibrary`, which is included in the default tuProlog engine, allows the Java world to be accessed dynamically from tuProlog in a simple and straightforward way. In order to maintain a clean separation between the two language paradigms, no notions of class and object are directly introduced in tuProlog. Instead, `JavaLibrary` allows Java classes and objects to be created, mapped upon Prolog terms, and used from tuProlog. Also, the mapping between Java and Prolog entities is strictly confined within the scope of the `JavaLibrary` constructs and operators – there and only there, some ground Prolog terms are interpreted as either Java classes or objects. Anywhere else, all we have are standard Prolog terms, with no special meaning at all.

Table 11 presents a `JavaLibrary` synopsis, providing the syntax and a short description of its constructs and predicates. In the remainder of this section, we first shortly discuss the main functions of such constructs (Section 4.1), then we provide the reader with a richer set of examples (Section 4.2).

4.1. *JavaLibrary constructs & predicates*

Generally speaking, `JavaLibrary` provides tuProlog with predicates to:

(1) create new Java objects;
(2) invoke both instance and static methods, and retrieve their results;
(3) select the public fields of an object or the public static fields of a class;
(4) cast objects to the specific Java types;
(5) perform dynamic compilation of Java classes;
(6) establish a priori bindings between Java objects and Prolog ground terms.

Table 11
Main `JavaLibrary` predicates

Functionality	Predicate(s)	Description

Object creation	`java_object(+ClassName, +ArgList, ?ObjRef)`
	
	Examples:
	`java_object('java.awt.Point', [2,3], P)`
	`java_object('java.lang.Integer', [303], n)`
		Creates a Java object of class *ClassName* by calling the constructor which matches the arguments specified in *ArgList*. If the predicates succeeds, *ObjRef* is bound to a term representing the object. *ObjRef* can be either a variable or a ground term.

Array creation	`java_object(+ClassName[], [+Len], ?ObjRef)`
	
	Example:
	`java_object('java.awt.Point'[], [100], V)`
		Specialises the `java_object/3` predicate for array creation.

Method invocation	*TargetRef* `<-` *MethodName*
	TargetRef `<-` *MethodName*`(+Arg0,+Arg1,...)`
	TargetRef `<-` *MethodName* `returns` *Res*
	TargetRef `<-` *MethodName*`(+Arg0,+Arg1,...)`
	`returns` *Res*
	Example 1:
	`java_object('java.awt.Point', [2,3], P),`
	`P <- getX returns X`
	Example 2:
	`Intclass = class('java.lang.Integer')`
	`Intclass <- parseInt('200') returns N`
		Invokes the method *MethodName* on the object associated to the *TargetRef* term, possibly passing arguments *Arg0*, *Arg1*, etc., and possibly binding the return argument to the *Res* term.
		To invoke static (class) methods, the compound term `class(`*ClassName*`)` should be used as *TargetRef*.

Field access	*TargetRef* `.` *FieldName* `<- set(+`*Arg*`)` *TargetRef* `.` *FieldName* `<- get(+`*Arg*`)`	Accesses the public field *FieldName* of object *TargetRef* to set/get its value to the value (or object reference) denoted by *Arg*. For static fields, the compound term `class(`*ClassName*`)` should be used as *TargetRef*.

Array access and management	`java_array_set(+`*ArrayRef*`, +`*Pos*`, +`*Content*`)`
	`java_array_get(+`*ArrayRef*`, +`*Pos*`, ?`*Content*`)`
	`java_array_length(+`*ArrayRef*`, ?`*Length*`)`
	
	Example:
	`java_object('java.awt.Point'[], [100], A),`
	`java_object('java.awt.Point', [1,2], Point),`
	`java_array_set(A, 0, Point)`
		Accesses position *Pos* of the array bound to the *ArrayRef* term to set/get the content of that position to the value (or object reference) associated to the *Content* term.
		The third predicate retrieves the array length and binds it to the *Length* term.

Cast	*Arg* `as` *TypeName*	Forces argument *Arg* to be considered of type *TypeName*.
Dynamic class compilation	`java_class(+`*Source*`, +`*ClassName*`,` `+`*PathList*`, ?`*ClassRef*`)`	Dynamically compiles the source text *Source* to define the new class named *ClassName*. *PathList* denotes the class path to be used for compilation. The compiled class, available as a `Class` instance, is associated to the *ClassRef* term.

Object creation — tuProlog allows for Java object creation and reference via the `java_object/3` predicate, which follows the syntax of Table 11. For instance, the following example

```
?- java_object('java.awt.Point', [2,3], p1), p1 <- getX returns X.
yes. X / 2.0
```

creates a new Java object of class `java.awt.Point` with initial configuration `[2,3]`, and maps it on the Prolog constant `p1`, which can then be used as a reference for the object within the subsequent subgoal. The lifetime of the above association is that of the current Prolog demonstration: so, when the demonstration ends, the association between `p1` and the Java object goes out of scope, and the `p1` term loses any special meaning.

For coherence with Prolog, the association between a Java object and the corresponding Prolog term has a single-assignment semantics: so, in the example below, if `p1` already refers to a Java object, it cannot be used again for a new object creation:

```
?- java_object('java.awt.Point', [2,3], p1),
   java_object('java.awt.Point', [4,3], p1).
no.
```

The subtle issue of whether the association between Prolog terms and Java objects should survive backtracking (privileging either efficiency or Prolog style and coherence) is left to the tuProlog designers and programmers, who can suitably set the `java_object_backtrackable` flag to either `true` or `false`.

Method invocation — tuProlog makes method invocation straightforward by mimicking the usual Java send-message schema via the `<-/2` and the (`<-`, `returns`)`/3` predicates, according to the quite intuitive syntax in Table 11 (see also the Swing example of Table 3 in Section 2.2). More precisely, the `<-/2` predicate, which neglects any return value, is thought for either void or boolean Java methods: in the latter case, it adopts the corresponding success/failure semantics. Instead, (`<-`, `returns`)`/3` predicate always succeeds, and binds the result of method invocation to the third argument as appropriate. In particular, Java types that can be mapped onto primitive Prolog data types (such as numbers or strings) are directly unified with the corresponding Prolog value, while Java objects are handled as object references in the same way of the first argument of `java_object/3`.

In the example seen above, for instance,

```
?- java_object('java.awt.Point', [2,3], p1), p1 <- getX returns X.
yes. X / 2.0
```

the Java method `getX` is invoked upon the newly-created Java object referred by term `p1` by using the homonymous Prolog constant `getX` as the second argument of the (`<-`,`returns`)`/3` predicate. Variable `X` is then bound to the value returned by the `getX` method.

Field selection — The ternary constructs (`.`, `<- set`)`/3` and (`.`, `<- get`)`/3` are used to access the public fields of an object or the public static fields of a class. According to the specification in Table 11, `set` and `get` are not methods of some class, but keywords of the ternary constructs: the former sets a public field to a specified value, the latter retrieves the field value.

Homologous constructs are provided by `JavaLibrary` for array access: in conjunction with method invocation operators, such predicates make it possible to compute over array elements, like in the example below:

```
?- java_object('java.awt.Point[]',[6], A),
   java_object('java.awt.Point',[1,2],P1), java_array_set(A,0,P1),
   java_object('java.awt.Point',[3,4],P2), java_array_set(A,1,P2),
   java_array_get(A,0,P), P <- toString returns S.
yes.   S / 'java.awt.Point[x=1,y=2]'
```

Method overloading resolution — To resolve a method invocation, tuProlog first looks for an exact match: if none is found, a compatible method is searched. For this purpose, the *is-assignable* semantics[3] is first considered: if it fails, compatibility via safe type conversion of primitive types (`int` → `long` → `double`, `int` → `float` → `double`) is tried. Nonetheless, it is sometimes necessary to explicitly cast arguments to match the method signature: this is what the `as` infix operator is for.

Dynamic compilation — tuProlog dynamic configurability also means support for *dynamic compilation* of Java classes, so that a new Java class can be created and loaded in tuProlog from a source text. Syntactically, the task is carried out by the `java_class/4` predicate, which takes the source text of the class to be compiled as its (string) argument, compiles it, and binds the result to a suitable instance of the Java meta-class `Class`. The predicate fails if the source contains errors, or the class cannot be located in the package hierarchy.

Predefined Object Bindings — Sometimes, it might be useful to pre-associate an existing Java object to a selected Prolog atom, so that the tuProlog VM can use it to reference the Java object directly: this is what the `JavaLibrary` `register` method is for.

```
boolean register(Struct objectRef, Object obj)
                 throws InvalidObjectIdException
```

The existing Java object *obj* is thus associated to the Prolog atom *objectRef*, expressed in Java as an `alice.tuprolog.Struct` instance: the result is `true` if everything is fine, `false` if the object *obj* is already registered under a different *objectRef*, or an exception if the latter term is not ground. Table 12 shows how to associate `System.out` to the Prolog atom `stdout`, so that a Prolog program can contain the sentence:

```
stdout <- println('What a nice message!')
```

with no need to bind the `stdout` atom to a Java object explicitly. Of course, it still remains true that, anywhere else in the tuProlog program, the `stdout` atom has no special meaning, and is interpreted as any other Prolog atom.

Finally, Java exceptions are indeed a critical aspect, too, that should be dealt with when mapping Java upon Prolog. Currently, given the mismatch between the two

[3] This semantics is defined by the `isAssignableFrom` method of the `Class` meta-class of the Java reflection API – please see Section 5 for details.

Table 12
Pre-association of the Java static object `System.out` to the Prolog atom `stdout`

```
Prolog engine = new Prolog();
Library lib = engine.getLibrary("alice.tuprolog.lib.JavaLibrary");
JavaLibrary jlib = (alice.tuprolog.lib.JavaLibrary) lib;
jlib.register(new Struct("stdout"), System.out);
```

exception models, Java exceptions are simply mapped into a tuProlog failure: even though conceptually correct, this prevents tuProlog from telling between 'real' failures and exceptions. Of course, this is not the end of the story: further research is in progress in order to define a semantically-sound approach for exception (and event) mapping between Java and tuProlog.

4.2. Examples

As a first example, let us consider the `Counter` class defined in Table 13, along with its tuProlog sample code. The first goal creates a `Counter` instance, called `aNiceCounter`, and binds it to the Prolog atom `myCounter`. This is then used to call `setValue(5)`, `inc`, and `getValue`, that retrieves the current counter value (hopefully, 6) and binds it to the Prolog variable `Value`, which is printed via the Prolog `write/1` built-in predicate. Of course, the call to `getValue` fails if `Value` is already bound to anything else than 6. The `name` public field of `myCounter` is finally accessed, printed, and set to the new value `NicerCounter`. The second goal demonstrates the invocation of a static method, calling `getVersion` and printing its string result via the `println(String)` method of the static object `System.out`. If an overloaded method is defined, the standard Java selection rules apply. So, if the `setValue(int)` method is uncommented, a call such as `setValue(5)` is bound to this method instead of the previous `setValue(long)`, since 5 is interpreted as an `int` constant. To have `setValue(long)` called, we should explicitly cast the argument to `long`, in quite the same way as we would do in a Java program, via tuProlog `as` operator.

Table 3 is another example of exploiting a standard Java API from tuProlog: the Swing `JFileChooser` object is bound to the Prolog variable `Dialog`, which is then used to invoke the required methods. Further similar examples using JDBC, RMI and CORBA can be found in [6].

Table 14 shows an example of dynamic compilation, where the Prolog variable `Source` is bound to an atom whose string representation is the source to be compiled. The `java_class/4` predicate dynamically compiles that source, taking as arguments (besides the source text) the class name (`Counter`) needed to locate the class in the package hierarchy, a list of class paths possibly required for making the compilation succeed (none in this case), and the Prolog atom (`counterClass`) to which the new `Class` instance is to be bound. Since the newly-compiled class is available as an instance of the `Class` meta-class, object creation must be performed indirectly, via the `newInstance` method; the new object, bound to the Prolog atom `myCounter`, is then used as desired – here, to set its value to 5, read it back, and print it on the screen. The dynamic compilation feature is particularly useful when combined with network access, allowing the text source of a class

Table 13
The `Counter` class declaration *(top)* and its use in tuProlog *(bottom)*

```
public class Counter {
    public String name;
    private long value = 0;
    public Counter() {}
    public Counter(String aName) { name = aName; }
    public void setValue(long val) { value = val; }
    // public void setValue(int val) { value = val; }
    public long getValue() { return value; }
    public void inc() { value++; }
    static public String getVersion() { return "1.0"; }
}
```

```
?- java_object('Counter',['aNiceCounter'],myCounter),
   myCounter <- setValue(5), % Variant: setValue(5 as long)
   myCounter <- inc,
   myCounter <- getValue returns Value, write(Value),
   myCounter.name <- get(MyName), write(MyName),
   myCounter.name <- set('NicerCounter').

?- class('Counter') <- getVersion returns Version,
   class('java.lang.System').out <- get(StdOut),
   StdOut <- println(Version).
```

Table 14
Predicate `java_class/4` performing dynamic compilation of Java code in tuProlog

```
test_dynamic_compilation:-
  Source = 'public class Counter { ... }', % see Table 13
  java_class(Source, 'Counter', [], counterClass),
  counterClass <- newInstance returns myCounter,
  myCounter <- setValue(5),
  myCounter <- getValue returns Value,
  write(Value).
```

to be downloaded and compiled on-the-fly: an example can be found in the tuProlog User Manual [25].

5. Implementation essentials

In order to achieve the key features of tuProlog – core minimality, openness, dynamic configurability, and straightforward Java integration –, the support for introspection, the ability to represent parts of a Java program as data, and the possibility of loading/unloading classes dynamically turned out to be crucial capabilities. This is why tuProlog implementation was rooted upon the *Java reflection* technology, which made it possible (i) to keep the core light-weight, allowing the built-ins to be defined outside the core, (ii) to achieve the required core openness and user-configurability, enabling built-ins to be discovered via introspection, and (iii) to access Java from tuProlog in the most dynamic way.

5.1. Dynamic extensibility and library support

tuProlog distinguishes the library load phase, the theory compilation phase, and the demonstration phase. In the first, a library is loaded or unloaded; in the second, a theory is compiled and the new built-ins are internally stored and bound; in the last, the new built-ins are eventually used.

Library load — The first goal here is to build a `Library` instance representing the new tuProlog library. This is easily done via the `forName` and `newInstance` methods of the Java reflection API:

```
Library lib = (Library) Class.forName(ClassName).newInstance()
```

where *ClassName* is obviously the name of the new library. The next step is to inspect the library methods, and look for those that adhere to the pattern defined in Section 3.3.1. More precisely, for a method to be the definition of a built-in, the following conditions must hold:

- the method name must end with `_N`, where *N* represents the built-in arity and therefore must exactly match the number of method arguments;
- the method return type must be either `boolean` (for predicates) or `Term` (for evaluable functors);
- each method argument must satisfy the *is-assignable* relationship (see Footnote 3) with respect to `Term` – that is, each argument type must be either `Term` itself, or one of its subclasses.

Again, the Java reflection API makes the job straightforward, in particular thanks to the `isAssignableFrom` method of the `Class` meta-class.

Theory compilation — When a theory is compiled, tuProlog verifies that each predicate defined by the theory is actually implemented by some library (it does so by consulting the hash table that stores all the recognised built-ins), and binds it to the corresponding library method. Technically, the `Struct` instance that represents the built-in is bound to a suitable `BuiltIn` instance, which encapsulates the reference to the implementing method. Conversely, when a library is eventually unloaded, its predicates and evaluable functors are removed from the hash table, and the link between the `Struct` and the `BuiltIn` object is unbound.

Demonstration — When a built-in is called, the tuProlog engine determines the actual parameters of the method to be invoked and invokes the method via reflection; on return, the method result is cast to the proper type (`boolean` for predicates, `Term` for evaluable functors), and made available as the predicate result. From the performance viewpoint, it is worth noting that, although reflection is widely used, class introspection (its heavier aspect) is used just once, in theory compilation: after that, the only run-time overhead is that of indirect method invocation.

5.2. *JavaLibrary* implementation

Three of the `JavaLibrary` fundamental functionalities – object creation, method invocation, and access to object fields – are based on Java reflection. Dynamic compilation, instead, makes no use of the Java reflection API, even though it still exploits `Class` meta-objects to represent classes.

Object creation — Object creation, performed via `java_object/3`, is heavily based on reflection: first, it is used to get the `Class` meta-object representing the provided *`ClassName`*, then to retrieve the actual arguments, identify the proper constructor, create the new instance, and bind it to the *`ObjectRef`* Prolog term. Retrieving the actual arguments, in particular, deserves special attention, since it implies the ability both to determine each argument class and value, and define a suitable Prolog mapping for the Java *null* reference. With respect to the latter issue, we explicitly excluded to map *null* onto a Prolog atom like `null` in order not to give a specific meaning to any Prolog atom: accordingly, we mapped *null* to the Prolog anonymous variable, both because this has already a special meaning in Prolog, and because its meaning recalls the same 'any value' idea that a *null* actual argument usually expresses in Java.

For each argument *`Arg`*, tuProlog checks whether it denotes the Prolog anonymous variable, an object, a primitive value, or the `as` type cast structure. In the first case, the argument value is obviously `null`, and its class is assumed to be `Object`: so, the argument class is set to the `Object.class` constant. If *`Arg`* denotes an object, the value is the object itself, and its class is obtained via the `getClass` inspection method. Instead, if *`Arg`* is a primitive value, the argument value is a newly-created instance of the corresponding wrapper class (`Integer`, `Double`, etc.), and its class is `Integer.TYPE`, `Double.TYPE`, etc., as appropriate. Finally, *`Arg`* can be an `X as Type` structure, which is handled quite similarly. If `Type` is primitive, the argument is wrapped as above: if, instead, it is the name of a class, the argument class is determined via the `forName` reflection method, while the value is either `X` itself (in the case of an existing object) or a newly-created number or string instance (in the case of a Prolog number or atom).

The third step is to choose the most appropriate constructor. To this end, we first search an exactly-matching constructor, via the `getConstructor` method of the reflection API; if none is found, all the public constructors are retrieved via introspection and filtered by a 'best-match' algorithm, which is based once again on the *isAssignableFrom* relationship – that is, each argument of the candidate constructor must be assignable from the actual argument type (possibly modulo safe type conversions). If, again, none is found, the object creation definitely fails, otherwise we select the *most specific* constructor: by definition, constructor *c1* is more specific than *c2* if, and only if, each argument *A2* of *c2 isAssignableFrom* the corresponding argument *A1* of constructor *c1*, or the two arguments are compatible (modulo safe type conversions).

Method invocation and access to object fields — Both the operators for invoking a method and accessing the public fields of an object or class are just syntactic sugar around one single basic mechanism, embedded in the `java_call_3` method:

```
public boolean java_call_3(Term objID, Struct method, Term resID)
```

where `method` is obviously the method to be invoked, and `resID` its result, if any. The behaviour of this method depends on the nature of the target entity `objID`, which may be either an unstructured alphanumeric term, or a structure like `class(C)` or `.(`*`ObjectRef`*`,`*`Field`*`)`. An unstructured term denotes that `objID` represents a reference to an object, `class(C)` indicates that the target is a class (i.e., a static method

is being invoked), while the latter corresponds to the access to a public field of an object or to a public static field of a class.

In the first case, we retrieve the associated object, determine its class via reflection, discover the value and type of the method arguments and finally look for a matching method, in the same way as for constructors above; the method is then invoked and its result (if it is non-void) is bound to `resID`.[4] Of course, `java_call_3` fails if the object cannot be found, or the required method does not exist. If, instead, `objID` is a `class(C)` structure, we use reflection to retrieve the class name, then we analyse the method arguments, look up for it, and invoke the static method quite like above. In the last case, `method` takes the form `set(`*`ContentID`*`)` or `get(`*`ContentID`*`)`, and `resID` is unused. We then check that *`ObjectRef`* denotes a valid object `o`, retrieve its class, and obtain a meta-representation of the target field as a `Field` meta-object, `f`. If *`ContentID`* denotes a non-primitive object `c`, we perform `field.set(o,c)` or `field.get(o)`, as appropriate – binding the `get` result to *`ContentID`*. Instead, if the field type is primitive, there is no object like `c`: so, the specific `Field` methods (such as `setInt`, etc.) are used instead of the generic `set` and `get`.

Dynamic compilation — Dynamic compilation exploits the standard `Compiler` class of the Java run-time (which usually refers to the JIT compiler): the provided source is compiled, and the resulting class immediately loaded into the tuProlog system.

6. Comparison with other approaches

The integration of (mainly functional and declarative) languages with ubiquitous languages, such as Java, is a primary issue for several approaches in the literature. Foreign Function Interface (FFI) is particularly relevant, especially in the context of mixed-language, component-based programming [1]. In tuProlog, reflection was exploited as the core mechanism for the FFI, and for dynamically extending the language with new libraries developed in Java. In the following we discuss those two aspects, by comparing tuProlog to two well-known, widely used systems: Jinni [22,24] – a Java-based Prolog system whose approach is not far from tuProlog one – and SICStus Prolog Jasper library [20] – a mainstream Prolog system not based on Java. Other systems will be shortly discussed in the Related Work section: namely, K-Prolog JIPL library [11] and Lambada [8] as Java-based Prolog systems, and Kawa [3,13], Jython [12], and MLj [2,15] as examples of Java-based, non-Prolog systems.

To compare both expressiveness and performance, we take two examples as our main references: the first tests the access to the Java world from Prolog (Tables 16 and 17), whereas the second stresses the issue of Prolog extension via new libraries exploiting Java (Tables 18–22). The latter example, in particular, is taken from the Jinni 2004 online documentation [28] and is then implemented also in tuProlog via JavaLibrary, in Jasper, and in tuProlog via an ad hoc library, so as to enable a full cross-comparison.

[4] If `objID` is a string constant, `method` necessarily belongs to class `String`, so the process is optimised, invoking the method directly.

Table 15
The Jasper integration with Java: the version of StringLibrary, with both the Java and Prolog side *(top)*, and creating and exploiting a Java Swing object *(bottom)*

```
import se.sics.jasper.*;

public class StringLibrary {
  static boolean toLowerCase(String source, SPTerm dest){
    String st = source.toLowerCase();
    try { dest.putString(st); } // putString is defined in SPTerm
    catch (ConversionFailedException e1) { return false; }
    catch (IllegalTermException e2) { return false; }
    return true;
  }
}
```

```
:- use_module(library(jasper)).

go(String, NewString, Result) :-
    jasper_initialize([classpath(['/my-jasper-examples'])], JVM),
    jasper_call(JVM,
        method('StringLibrary', 'toLowerCase', [static]),
        lib_to_lower_case(+string, -term, [-boolean]),
        lib_to_lower_case(String, NewString, Result) ).
```

```
:- use_module(library(jasper)).

choose_file(File) :-
   jasper_initialize([classpath(['/my-jasper-examples'])], JVM),
   jasper_new_object(JVM,
     'javax/swing/JFileChooser', init, init, Dialog),
   jasper_null(JVM, NullParent),
   jasper_call(JVM,
     method('javax/swing/JFileChooser','showOpenDialog',[instance]),
     show_open_dialog(+object('javax/swing/JFileChooser'),
                      +object('java/awt/Component'), [-integer]),
     show_open_dialog(Dialog, NullParent, Result)
   ),
   jasper_call(JVM,
     method('javax/swing/JFileChooser','getSelectedFile',[instance]),
     get_selected_file(+object('javax/swing/JFileChooser'),
                       [-object('java/io/File')]),
     get_selected_file(Dialog, File)
   ).
```

6.1. Jinni

Jinni (Java INference and Networked Interactor) is a Prolog system with object and agent-oriented extensions, available for the Java and .NET platforms [22,24,28]. Jinni features a light-weight and multi-threaded compiler, and is intended to be used as a flexible scripting tool for gluing together knowledge processors with Java and .NET components in distributed applications.

Table 16
Accessing Java from Prolog: tuProlog *(top)* and Jasper *(bottom)*. See also Table 17

```
loop(0,_,_):- !.
loop(N,List,Rnd):-
  java_call(Rnd,nextInt,X),java_call(Rnd,nextInt,Y),
  java_object('java.awt.Point', [X,Y], Obj),
  java_call(List,add(Obj),_),
  N1 is N-1, loop(N1, List, Rnd).
test(N) :-
  class('java.lang.System') <- currentTimeMillis returns T0,
  java_object('java.util.ArrayList', [], List),
  java_object('java.util.Random', [], Rnd),
  loop(N, List, Rnd),
  class('java.lang.System') <- currentTimeMillis returns T1,
  DT is T1-T0, stdout <- println(DT).
```

```
:- use_module(library(jasper)).

loop(0,_,_,_):- !.
loop(N,List,Rnd,JVM):-
  jasper_call(JVM, method('java/util/Random', 'nextInt',[instance]),
    next_int(+object('java/util/Random'), [-integer]),
    next_int(Rnd,X)),
  jasper_call(JVM, method('java/util/Random','nextInt', [instance]),
    next_int(+object('java/util/Random'), [-integer]),
    next_int(Rnd,Y)),
  jasper_new_object(JVM, 'java/awt/Point',
    init(+integer,+integer), init(X,Y), Obj),
  jasper_call(JVM, method('java/util/ArrayList', 'add', [instance]),
    add(+object('java/util/ArrayList'),
        +object('java/lang/Object'), [-boolean]),
    add(List,Obj,Res)),
  N1 is N-1, loop(N1,List,Rnd,JVM).
test(N) :-
  jasper_initialize([], JVM),
  jasper_call(JVM, method('java/lang/System', 'currentTimeMillis',
                  [static]),
    current_time_millis([-long]),current_time_millis(T0)),
  jasper_new_object(JVM,'java/util/ArrayList', init, init, List),
  jasper_new_object(JVM,'java/util/Random', init, init, Rnd),
  loop(N, List, Rnd, JVM),
  jasper_call(JVM, method('java/lang/System', currentTimeMillis',
                  [static]),
    current_time_millis([-long]), current_time_millis(T1)),
  DT is T1-T0, write(DT), nl.
```

Generally speaking, Jinni provides a wider and deeper model for the integration between the logic and object-oriented paradigms with respect to tuProlog: in fact, it supports both the definition of new Java classes from the Prolog environment, and constructs to work directly with objects and state in its extended logic environment. tuProlog intentionally

Table 17
Accessing Java from Prolog (see also Table 16): Jinni *(top)* and performance comparison *(bottom)* for the query `test(N)`: timings are in seconds

```
loop(0,_,_):-!.
loop(N,List,Rnd):-
  invoke_java_method(Rnd,nextInt,X), invoke_java_method(Rnd,nextInt,Y),
  new_java_object('java.awt.Point'(X,Y),Obj),
  invoke_java_method(List,add(Obj),_),
  N1 is N-1, loop(N1,List,Rnd).

test(N):-
  ctime(T0),
  new_java_object('java.util.ArrayList',List),
  new_java_object('java.util.Random',Rnd),
  loop(N,List,Rnd),
  ctime(T1), DT is T1-T0,
  new_java_class('java.lang.System',S), get_java_field(S,out,Stdout),
  invoke_java_method(Stdout,println(DT),_).
```

N (#iterations)	tuProlog	Jasper	Jinni
100	0.07	0.03	0.06
1000	0.29	0.26	0.30
10000	2.94	2.55	3.21
50000	15.58	13.02	16.15
100000	31.88	25.91	32.32

does not provide such features, in order to keep the two linguistic paradigms clearly separate: this is why Java resources can be accessed only by means of a special library, and no ad hoc mechanisms are introduced in the logic core. Apart from that, both Jinni and tuProlog enable Java resources to be accessed from the logic environment (via special predicates in Jinni, via `JavaLibrary` predicates in tuProlog), and, conversely, Prolog engines to be exploited from Java, via their own APIs. In particular, both systems exploit Java reflection to have their predicates create objects, invoke methods and access public fields. Another fundamental feature of tuProlog is the dynamic extensibility of the language by means of libraries written in Java, which can define new predicates and functors. In Jinni, this form of extensibility is meant to be achieved through the direct exploitation of Java resources from the Prolog environment, with no need of new libraries. The reference case study (Table 17) is the development of a library to create hash dictionaries, enabling the insertion and removal of data elements hashed according to a specific key.

6.2. Jasper

Jasper is the SICStus Prolog [20] bidirectional interface towards Java. Since SICStus is not Java-based, Jasper exploits the Java Native Interface (JNI) to access the JVM packaged in a dynamically-loaded library (such as a Windows DLL or a Unix `.so` shared object).

Table 18
Testing tuProlog extension via JavaLibrary (see also Tables 19–22)

```
hashtable:-
  java_object('java.util.HashMap',[],Map),
  abolish (map(_)), assert(map(Map)).
put_data(Key,Data):-
  map(Map), java_call(Map, put(Key,Data),_).
get_data(Key,Data):-
  map(Map), java_call(Map, get(Key),Data).
remove_data(Key):-
  map(Map), java_call(Map, remove(Key),_).
loop(0):-!.
loop(N):-
  java_object_('java.lang.Integer',[N],Data),
  java_call(Data,toString,Key),
  put_data(Key,Data), get_data(Key,_),remove_data(Key),
  N1 is N-1, loop(N1).

test(N):-
  hashtable,
  class('java.lang.System') <- currentTimeMillis returns T0,
  loop(N),
  class('java.lang.System') <- currentTimeMillis returns T1,
  DT is T1-T0, nl, write(DT), nl.
```

On the Java side, a set of classes represents the SICStus runtime system and Prolog data types (such as `SICStus`, `SPTerm`, etc.). On the Prolog side, a library module makes it possible not only to create objects, invoke methods, and manage object references, but also to control the loading and unloading of the JVM via the `jasper_initialize` and `jasper_deinitialize` predicates. The drawback is that the burden of handling such details is upon users, who must explicitly create a reference to a JVM instance in their Prolog program and pass it along to all the involved predicates. More generally, Jasper users are required to be aware of several system-level details, ranging from the lifetime of object references when using predicates like `jasper_create_global_ref/3`, `jasper_delete_global_ref/2`, `jasper_delete_local_ref/2`, to the different effects on foreign resources when loading a Prolog saved state via Java methods like `restore` and `load`, to possible memory leaks when managing Prolog terms (`SPTerms`) from Java, up to the need of specifying the position of some key library files (such as `jvm.dll` or `spnative.dll` on a Windows system) in the operating system path for Jasper to work – especially if the parent application is on the Java side.

Like tuProlog, Jasper enables its users both to access and handle Java resource from Prolog, and to instantiate and control SICStus engines from Java; in Jasper, callbacks are also possible, the level of nesting being limited only by the available memory. However, since minimality was not among SICStus concerns, each instantiation of the `SICStus` class corresponds to the activation of one independent copy of the SICStus run-time, which is admittedly "a rather heavy-weight entity" [21, Ch. 10]. Moreover, since SICStus is designed to be used mainly as an independent production system rather than a component

Table 19
Testing Prolog extension via Java in Jasper (see also Table 18, 20–22)

```
:- use_module(library(jasper)).

hashtable(JVM):-
  jasper_new_object(JVM, 'java/util HashMap', init, init, Map),
  retractall(map(_)), assert(map(M)).
put_data(JVM,Key,Data):-
  map(Map),
  jasper_call(JVM,
    method('java/util/HashMap','put',[instance]),
    put(+object('java/util/HashMap'), +object('java/lang/Object'),
        +object('java/lang/Object'), [-object('java/lang/Object')]),
    put(Key,Data,_)).
get_data(JVM,Key,Data):-
  map(Map),
  jasper_call(JVM,
    method('java/util/HashMap','get',[instance]),
    get(+object('java/util/HashMap'), +object('java/lang/Object'),
        [-object('java/lang/Object')]),
    get(Map,Key,Data)).
remove_data(JVM,Key,Data):-
  map(Map),
  jasper_call(JVM,
    method('java/util/HashMap','remove',[instance]),
    remove(+object('java/util/HashMap'), +object('java/lang/Object'),
           [-object('java/lang/Object')]),
    remove(Map,Key,_)).
loop(_,0):-!.
loop(JVM,N):-
  jasper_new_object(JVM,
    'java/lang/Integer', init(+integer), init(N), Obj),
  jasper_call(JVM, method('java/lang/Integer','toString',[instance]),
    to_string(+object('java/lang/Integer'),
              [-object('java/lang/String')]), to_string(Obj,S)),
  put_data(JVM,S,Obj), get_data(JVM,S,_), remove_data(JVM,S,_),
  N1 is N-1, loop(JVM,N1).

test(N):-
  jasper_initialize([],JVM), hashtable(JVM),
  jasper_call(JVM,method('java/lang/System','currentTimeMillis',[static]),
    current_time_millis([-long]),current_time_millis(T0)),
  loop(JVM,N),
  jasper_call(JVM,method('java/lang/System','currentTimeMillis',[static]),
    current_time_millis([-long]),current_time_millis(T1)),
  DT is T1-T0, nl, write(DT), nl.
```

in a dynamic environment, the Prolog program to be loaded from the Java side must be retrieved from a SICStus saved state, via the `restore` method: so, Jasper support for dynamic configurability is quite limited.

Table 20
Testing Prolog extension via Java in Jinni (see also Tables 18, 19, 21, 22)

```
hashtable:-
  new_java_class('java.util.HashMap',C),
  new_java_object(C,void,JavaHashTable),
  object<=JavaHashTable.
put_data(Key,Data):-
  object=>T, invoke_java_method(T,put(Key,Data),_).
get_data(Key,Data):-
  object=>T, invoke_java_method(T,get(Key),Data).
remove_data(Key):-
  object=>T, invoke_java_method(T,remove(Key),_).
loop(0):-!.
loop(N):-
  new_java_object('java.util.Integer'(N),Data),
  invoke_java_method(Data,toString,Key),
  put_data(Key,Data),get_data(Key,_), remove_data(Key),
  N1 is N-1, loop(N1).

test(N):-
  hashtable,
  ctime(T0), loop(N), ctime(T1),
  DT is T1-T0, nl, write(DT), nl.
```

With respect to the multi-paradigm integration, it should first be noted that Jasper does not provide a real mapping on the Java side for the concepts of Prolog variable and Prolog goal: in fact, Prolog variables are transposed on the Java side only as names associated to a binding expressed as an `SPTerm` instance by means of a standard Java `Map` object. Analogously, a goal in a Jasper query is just a conventional Java string. The Jasper `Query` interface, instead, is similar to the tuProlog `Prolog` class, and provides methods to open a query, get a solution, and retrieve further solutions: so, a Prolog top-level in Jasper [21, Ch. 10] is not too different from the tuProlog console-based interpreter (Table 6), although the Jasper choices discussed above make it a little more complex. Moreover, Jasper also provides query methods with a 'cut-fail' semantics, as well as a means to handle events and exceptions between Prolog and Java – an issue which is planned to be addressed only in tuProlog 2.0.

From the expressiveness viewpoint, apart from Jasper's need to load a JVM instance explicitly, both Jasper and tuProlog support dynamic object creation and method invocation, including static class methods. However, the `jasper_call` primitive requires the user to specify the method to be called in a rather intricate way – first, the method class, name, and qualifiers; then, its signature, expressed by a somehow unnatural convention; finally, the actual arguments – in contrast to the tuProlog complete absence of pre-declarations. Moreover, the above-cited Jasper type convention is currently unable to express array types. Jasper Java/Prolog mapping is also a little complex: in particular, Prolog terms are mapped onto `SPTerm` instances, whose class closely corresponds to

Table 21
Testing tuProlog extension via ad hoc libraries (see also Tables 18–20, 22)

```
package alice.test.tuprolog;
import alice.tuprolog.*;
import java.util.*;

public class HashLibrary extends Library {
    private HashMap dict;
    public boolean hashtable_0(){
        dict = new HashMap();
        return true;
    }
    public boolean put_data_2(Term key, Term obj){
        dict.put(key.toString(),obj);
        return true;
    }
    public boolean get_data_2(Term key, Term res){
        Term result = (Term)dict.get(key.toString());
        return unify(res,result);
    }
    public boolean remove_data_1(Struct key){
        dict.remove(key.toString());
        return true;
    }
}
```

```
:- load_library('alice.test.tuprolog.HashLibrary').

loop(0):-!.
loop(N):-
  java_object('java.lang.Integer',[N],Data), Key=N,
  put_data(Key,Data), get_data(Key,_), remove_data(Key),
  N1 is N-1,
  loop(N1).

test(N):-
  hashtable,
  class('java.lang.System') <- currentTimeMillis returns T0,
  loop(N),
  class('java.lang.System') <- currentTimeMillis returns T1,
  DT is T1-T0,
  nl, write(DT), nl.
```

SICStus' foreign interface for the C language.[5] Finally, Jasper notation for fully-qualified Java class names does not follow the Java standard: rather, it is inspired by the JVM internal naming scheme (package and class names are separated by '/' rather than '.').

[5] Actually, the SICStus manual does not present the `SPTerm` class in detail: the available methods have to be inferred from the FFI for the C language.

Table 22
Performance comparison for the query `test(N)` (timing in seconds)

N (#iterations)	tuProlog	Jasper	Jinni	tuProlog + HashLibrary
100	0.07	0.05	0.07	0.03
1000	0.53	0.44	0.48	0.10
10000	5.47	4.43	5.41	1.07
50000	29.11	22.70	26.50	5.76
100000	60.92	45.51	48.64	12.08

For the sake of concreteness, Table 15 (top) presents the Jasper version of the tuProlog `StringLibrary` shown in Table 2. While the Java side is quite straightforward and similar to the tuProlog case, the Jasper/Prolog side clearly outlines the information required by Jasper to call a Java method: the first of `jasper_call` arguments – a `method/3` term – specifies the class name, the method name and qualifier (static vs. instance method), the second argument provides the method signature, while the third specifies the actual parameters. Both last arguments are expressed as a Prolog term whose name must mimic, adopting the Prolog conventions, the name of the corresponding Java method, which is supposed to follow the standard Java conventions: so, for instance, the `toLowerCase` method translates as the `to_lower_case` predicate. It is the user's responsibility to guarantee that the above conventions are respected. For what concerns the specification of the method signature, the `to_lower_case` term embeds a term (in the form `+`*`type`* or `-`*`type`*) for each method argument, plus possibly one more analogous term between square brackets for the method return type (which may be lacking if the method is void). As a further example, Table 15 (bottom) presents the Jasper version of the file chooser example shown in Table 3. The interesting part here is how `jasper_new_object` specifies the instance creation: in particular, the first occurrence of `init` represents the constructor signature (in general, it would be something like `init(`*`formalArgumentList`*`)`, with each argument being specified according to the above syntax), while the second occurrence supplies the actual parameters (here, none; in general, something like `init(`*`actualArgumentList`*`)`). The Prolog variable `Dialog` embeds the reference to the newly-created object, and is later exploited to invoke instance methods.

6.3. Overall comparison and benchmarks

Unlike Jasper, both 2P and Jinni aim to provide an agile and lightweight support for Java/Prolog bidirectional integration; in particular, both make such a support available also as a pure Java component, so that it can be used for the development of complex Java applications (such as agent-based applications, coordination and mobility infrastructures, etc.). The above deployability and lightness features are necessarily reduced in Jasper, which is based on the (efficient) SICStus runtime rather than on Java. tuProlog also provides a direct support for language extensibility, by allowing new built-in libraries to be developed in Java; Jasper and Jinni support that feature indirectly, by providing access to Java resources.

With respect to performance, it should first be observed that efficiency was not a primary issue for tuProlog, which was not designed to be a commercial product; Jinni and SICStus, instead, are well-known and efficient commercial systems, leaders in their application domain (i.e., Java-based and non-Java Prolog systems, respectively). However, the benchmarks indicate that tuProlog performance in Java/Prolog integration is very much like the two other approaches. Quite interestingly, tuProlog performance significantly improves when exploiting its capability to link libraries developed in Java.

The first benchmark (Tables 16 and 17) concerns the access and use of Java resources from the Prolog environment, avoiding system-specific issues (e.g. special handling of primitive types), and focusing on iterated object creation, method invocation, and field access: the test creates a Java list and fills it with N Java AWT points with random coordinates. The second benchmark (Tables 18–22) concerns the development and use of a new library, defining a theory to create hash dictionaries, insert data elements, and later remove them. The test gives an idea how effectively the language can be extended. All tests were performed on a PowerMac G4 800 Mhz 1 Gb RAM, with Mac OS X 10.3.5 and JVM 1.4.2; the Prolog systems were Jinni 2004, tuProlog 1.2.1 and SICStus Prolog (Jasper) 3.11.2. Each test was executed several times for each approach, and the lowest timing results were considered.

The first test underlines the similarities between tuProlog and Jinni with respect both to the agility and seamlessness in accessing the Java world, and to the related performance, based on Java reflection. Interestingly, their performance is not too far from Jasper, although the latter is compiled for the target platform rather than for the JVM[6]; however, Jasper overhead for pre-declarations when creating objects and invoking methods is significant. The second test focuses on the extendibility of the declarative language by means of a library developed in Java. In both Jinni and Jasper, developing such a library means to write a simple theory, whose rules exploit `java.util.HashMap` objects; in tuProlog, the problem can be faced in two ways – either (like Jinni and Jasper) by writing a theory that accesses Java resources via JavaLibrary (Table 18), or by defining in Java a specific tuProlog library (Table 21).

The two possible approaches available in tuProlog refer to two conceptually different situations: one simply exploits a general mechanism to access Java resources and existing libraries, while the other means to *extend the language*, without forcing users to adopt a different programming paradigm. In the latter case, Prolog libraries are perceived by users as collections of new built-in predicates and functors, possibly provided with a Prolog theory: so, there is no need for the Prolog user to be aware of any object-oriented construct, as well as of the Java world in general. In the former case, instead, awareness of Java constructs is required. Quite interestingly, the latter approach seems to perform better than the former (see Table 22): in fact, although the new built-ins are loaded dynamically, invoking a library predicate turns out to be cheaper than invoking a method on a Java object, since it requires fewer reflection steps (method lookup is avoided).

[6] Obviously, the more the test includes Prolog code not accessing Java, the better Jasper performs, since it is compiled directly for the target platform.

7. Related work and conclusions

Several Java-related and Java-based interpreters/compilers for foreign languages can be found in the literature and on the Web [23]. Like tuProlog, some provide an API to exploit the foreign language (mainly functional or declarative) from Java, and, conversely, to access Java resources via a reflection-based FFI: examples are Jython [12] and Kawa [13]. The first is a Java-based open-source interpreter for the Python language [12], considered well suited for embedded scripting, interactive experimentation, and rapid application development. Jython provides a bidirection integration with Java, and represents the Python basic data types as a hierarchy of Java classes. On the other hand, Kawa FFI is quite similar to tuProlog and Jinni.

Among non-Java-based systems, K-Prolog JIPL [11] is a Prolog library whose approach to Java access is similar to Jasper. Other non-Prolog systems, such as Lambada [8] (a framework for inter-operability between Java and the Haskell functional language) and Haskell systems in general [19,9] also exploit the JNI to support Java integration. CIAO Prolog [4], instead, follows the idea of spawning a JVM as a stand-alone process, interacting with Prolog via sockets. These choices were inadequate to address tuProlog key requirements of minimality, dynamic configurability and easy deployability: in particular, the second approach based on a separate Java process is too complex, requiring a larger amount of resources and a more careful setup. In contrast, tuProlog setup is straightforward, and its core package requires little resources, its size being only 200 KB (`JavaLibrary` alone is just 30 KB).

Further approaches, such as MLj [15] (a compiler producing Java bytecode from Standard ML [2]), Haskell [27], and functional languages [14], compile a foreign language source into JVM bytecode. Of course, applications and libraries developed in the foreign language and compiled to bytecode are faster, since Java operations are often statically resolved and inlined, avoiding reflection. Apart from the extra resources to produce optimised bytecode, this approach requires off-line processing, so it seems unsuitable for the dynamic and interactive environments for which tuProlog is intended.

In short, tuProlog enables both the features of a standard Prolog system to be brought to the Java context, and vice versa. This makes it possible to exploit the power of symbolic reasoning within Java applications, and, conversely, to exploit the wide collection of Java resources in a Prolog environment "as is". In addition, it also supports extensibility of the Prolog-based language with special-purpose Prolog libraries efficiently written in Java. Clearly, the benefits of a multi-paradigm approach are strictly dependent on the integration model: in tuProlog, reflection techniques allowed an effective form of bidirectional integration to be realised without complex tricky mechanisms, keeping the core minimal yet dynamically configurable.

From the performance viewpoint, today software engineering techniques mostly aim to address complexity through expressiveness and adequate abstractions, rather than chasing efficiency and optimisation; in particular, efficiency is not a primary issue in Internet applications when compared to adaptability, support for distribution, heterogeneity, openness, and unpredictability. Nevertheless, a careful balance between the use of reflection at the interface between core and libraries, and the use of pre-compiled information acquired at theory/library load time makes tuProlog performance comparable

to other systems. tuProlog works as the basic brick of the TuCSoN [17] and LuCe [5] agent infrastructures, as well as as a key component in both academic and industrial projects. Current work is devoted to model exception and event handling inside `JavaLibrary`, taking the ISO standard as the main reference, so as to enable the definition of listeners for Java events in the form of tuProlog theories.

References

[1] N. Benton, A. Kennedy, Interlanguage working without tears: Blending SML with Java, ACM SIGPLAN Notices 34 (9) (1999) 126–137.
[2] N. Benton, A. Kennedy, G. Russell, Compiling Standard ML to Java bytecodes, ACM SIGPLAN Notices 34 (1) (1999) 129–140.
[3] P. Bothner, Kawa: Compiling Scheme to Java, in: LISP Users Conference: LISP in the Mainstream (40th Anniversary of LISP), Berkeley, CA, USA, 1998.
[4] The CIAO Prolog development system WWW site, http://www.clip.dia.fi.upm.es/Software/Ciao.
[5] E. Denti, A. Omicini, LuCe: a tuple-based coordination infrastructure for Prolog and Java agents, Autonomous Agents and Multi-Agent Systems 4 (1–2) (2001) 139–141.
[6] E. Denti, A. Omicini, A. Ricci, tuProlog: A light-weight Prolog for Internet applications and infrastructures, in: Ramakrishnan [18], pp. 184–198.
[7] P. Deransart, A. Ed-Dbali, L. Cervoni, Prolog: The Standard, Springer, 1996.
[8] S. Finne, E. Meijer, Lambada, Haskell as a better Java, Electronic Notes in Theoretical Computer Science 41 (1) (2000).
[9] S. Finne, E. Meijer, D. Leijen, S. Peyton Jones, Calling hell from heaven and heaven from hell, in: International Conference on Functional Programming, ICFP'99, Paris, France, 1999, pp. 114–125.
[10] A. Igarashi, M. Viroli, On variance-based subtyping for parametric types, in: ECOOP 2002 — Object-Oriented Programming, LNCS, vol. 2347, Springer-Verlag, 2002, pp. 441–469.
[11] JIPL: Java interface to Prolog, http://www.kprolog.com/jipl/.
[12] Jython home page, http://www.jython.org/.
[13] Kawa, the Java-based Scheme system, http://www.gnu.org/software/kawa/.
[14] G. Meehan, M. Joy, Compiling lazy functional programs to Java bytecode, Software Practice and Experience 29 (7) (1999) 617–645.
[15] MLj home page, http://www.dcs.ed.ac.uk/home/mlj/.
[16] A. Omicini, A. Natali, Object-oriented computations in logic programming, in: M. Tokoro, R. Pareschi (Eds.), Object-Oriented Programming, LNCS, vol. 821, Springer, 1994, pp. 194–212.
[17] A. Omicini, F. Zambonelli, Coordination for Internet application development, Autonomous Agents and Multi-Agent Systems 2 (3) (1999) 251–269.
[18] I. Ramakrishnan, Practical Aspects of Declarative Languages, LNCS, vol. 1990, Springer, 2001.
[19] C. Reinke, Towards a Haskell/Java connection, in: K. Hammond, T. Davie, C. Clack (Eds.), Implementation of Functional Languages, LNCS, vol. 1595, Springer, 1999, pp. 200–215.
[20] SICStus Prolog home page, http://www.sics.se/ps/sicstus.html.
[21] SICStus Prolog User's Manual, 2004.
[22] P. Tarau, Jinni: a lightweight Java-based logic engine for Internet programming, in: K. Sagonas (Ed.), International Workshop on Implementation Technologies for Programming Languages based on Logic, JICSLP'98, Manchester, UK, 1998, pp. 1–15.
[23] R. Tolksdorf, Programming languages for the Java virtual machine, http://www.robert-tolksdorf.de/vmlanguages.html.
[24] S. Tyagi, P. Tarau, A most specific method finding algorithm for reflection based dynamic Prolog-to-Java interfaces, in: Ramakrishnan [18], pp. 322–336.
[25] tuProlog home page, http://lia.deis.unibo.it/research/2P/.
[26] tuProlog at SourceForge, http://tuprolog.sourceforge.net.
[27] D. Wakeling, Mobile Haskell: Compiling lazy functional programs for the Java virtual machine, in: C. Palamidessi, H. Glaser, K. Meinke (Eds.), Principles of Declarative Programming, LNCS, vol. 1490, Springer, 1998, pp. 335–352.
[28] The world of Jinni, http://www.binnetcorp.com/Jinni/index.html.