PolimiRLToolkit

An introduction

Version 1.0

May 15, 2006

Contents

1 What is PRLT? 3

2 PRLT installation 3

2.1 System requirements 3

2.2 Download sources 4

2.3 Installation 4

3 Using PRLT 6

3.1 How to run experiments 6

3.2 How to use the log system 6

4 Understanding PRLT 9

4.1 Architecture overiew 9

4.2 Directory tree 12

5 XML, XSD and experiments descriptions 14

5.1 XML data parsing 16

5.2 XSD in PRLT 18

A Savane CVS How to 21

B Simple CVS How to 22

C Compiling MySQL++ 23

C.0.1 MySQL++ 24

D Eclipse 25

E Gnuplot essentials 27

F Valgrind 28

G Callgrind and Kcachegrind 28

H Useful links 29

I Acknowledgment 30

1 What is PRLT?

PolimiRLToolkit (Polimi Reinforcement Learning Toolkit[1] aka PRLT) is a GNU GPL framework developped by Politecnico di Milano, to create and test reinforcement learning alorithms.

We can also find some other framework with the same intent. One of the most used framweork for RL (*R*einforcement *L*earning) is the toolkit of the University of Alberta [2] that aims at helping who wants to learn, teach and use reinforcement learning. A big amount of its core and environments is written in c, but you can find also lisp and tk. A strong point is the huge number of interfaces to others language that is offered by this toolkit.
Another important feature of this software is given by the RL-GLue[3]: "a standard interface for interconnecting reinforcement learning agents (controllers) and RL environments", used to help creating a standard for benchmark so that you can compare your result with the those coming from other sources.

From Graz University comes the Reinforcement Learning Toolbox[4], too. This is a good work containing interfaces for learning from other controllers, many kind of algorithms (TD-lambda Q-Learning learning, TD-Lambda V-Learning, Actor critic learning, Advantage Learning, model based reinforcement learning, policy search algorithm and VAPS.), various way to build your q-function, semi MDP, hierarchical RL, logging and error recognition tools.

You can find more informations about the development of reinforcement learning at Alberta on the page http://rlai.cs.ualberta.ca/RLAI/rlai.html.

There are also others reinforcement learning toolkit with different characteristics, but the difference of PolimiRLToolkit from those is that it wants to offer not only classic algorithms and environments, but also multiagent algorithms, interfaces to other applications (3D simulator, java programs and RL-benchmark), a convenient way to describe experiments in XML, log in MySQL and so on.

2 PRLT installation

2.1 System requirements

PolimiRLToolkit is written for linux, mostly in c++, has java interfaces to others programs, and environments descriptions are in XML. It can also use XercesC, MySQL, Octave, flex and bison.

[1] https://savane.elet.polimi.it/projects/prlt/
[2] http://rlai.cs.ualberta.ca/RLAI/RLtoolkit/RLtoolkit1.0.html
[3] http://rlai.cs.ualberta.ca/RLBB/RL-Framework-concepts.html
[4] http://www.igi.tugraz.at/ril-toolbox/general/overview.html

PolimiRLToolkit has been tested on Mandriva, Kubuntu, Ubuntu e Gentoo and needs:

- Xerces (es. libxerces-c26) (for reading configuration files)
- bison (for parsing)
- flex (for parsing)

Optional, but very useful

- MySQL++ >= 2.0.4: for loggin
- Octave: as a basis for the management of symmetries
- Gnuplot: very useful to graph experiments results
- CVS: to get the development version

2.2 Download sources

The site project is https://savane.elet.polimi.it/projects/prlt/. From there you can download the project tarball. Another way to obtain sources is to bring them directly from CVS as described in appendix A

2.3 Installation

Once you have downloaded PolimiRLToolkit package in $INST_DIR

1. install Xerces if you already don't have it. Most distros provides it in an easy to install format. All you need is:
   - libxerces-c26
   - libxerces-c26
   - xerces-c
2. unpack PolimiRLToolkit :

```
cd $(INST_DIR)
tar xzf root.tar.gz
```

3. Optional: install MySQL if you alread don't have it. You can also avoid install MySQL and MySQL++, so if you want you can skip this and next two points.

4. install MySQL++.
   If your distro doesn't provides it, you can download it directly from MySQL++ site `http://tangentsoft.net/mysql++/` and install it. RPM based distro's users beware to read their specific paragraph on the site.

5. patch MySQL++:

   - enter the config directory:

     ```
     cd $(INST_DIR)/prlt/core/config
     ```

   - being superuser...:

     ```
     su
     ```

   - ... copy custom.h and custom-macros.h in the MySQL++ installation directory. This is, for mandriva and gentoo:

     ```
     mv /usr/include/mysql++/custom.h /usr/include/
        mysql++/custom.h.old
     mv /usr/include/mysql++/custom-macros.h /usr/
        include/mysql++/custom-macros.h.old
     cp custom* /usr/include/mysql++/
     ```

6. configure your environment setting some variables in ∼/.bashrc according to your system configuration. In order to do this add this lines to your ∼/.bashrc:

   ```
   export XERCESINCLUDES=/usr/include/xercesc/
   export MYSQLINCLUDES=/usr/include/mysql
   export MYSQLPPINCLUDES=/usr/include/mysql++
   export MYSQLLIB=/usr/lib/mysql
   ```

   exit this shell and start a new one, so that new environments variables can be updated.

7. compile `PolimiRLToolkit` :

   ```
   cd $(INST_DIR)/prlt/core/src/
   make tests
   ```

   if you have problems like the error

   ```
   libmysqlpp.so.2 not found during the execution
   ```

   or with any kind of library, you should modify your ∼/.bashrc to set the environment variable `$LD_LIBRARY_PATH`:

```
export LD_LIBRARY_PATH=/usr/local/lib
```

...or `/usr/lib`, according to your system configuration.

Now, even if you don't have a real log system, you can try some simple experiments: in core/data/experiment you can find ready to use experiments as described in section 3.1, ordered according to each implemented algorithm.

3 Using PRLT

3.1 How to run experiments

`PolimiRLToolkit` provides some ready to use experiments, just to test it: the directory prlt/core/data/experiment contains the directory benchmark, templates, and a list of directories corresponding to some learning algorithms. Entering one of them, you will find a list of XML files, each of them contains parameters required by the learning algorithm like the number of trials, the learning rate and so on. All you have to do is to run an experiment simply entering the directory prlt/core/test and call the executable RLtoolkit passing the chosen XML file:

```
cd $(INST_DIR)/prlt/core/test/
make tests
./toolkit -n ../data/experiment/qlearning/GridWorld.xml
```

The output will depend on the chosen experiment.

3.2 How to use the log system

Now that you can run some experiment, you may want to have additional informations about the learning process. This means that you need a way to generate a log of your experiment.

`PolimiRLToolkit` helps the user to create a log thanks to the classes LoggerManager and Logger. This provides a common interface to the programmer, so that he can log the experiment on an XML, a database or to a text file. Following this porpuse we have three types of loggers that implements this common interface: a TextLogger, an XMLLogger and a DBLogger. Every base core class has a pointer to an object of type Logger implementing one of the classes said before. If you need to log something, all you have to do is to use that logger: open a context, log and then close the context. Depending of the type of logger you are using, datas will be written to an XML, a text file or to a MySQL database.

Let analize the log obtained by the experiment in the directory prlt/core/data/experiment/qlearning/GridWorld.xml. `PolimiRLToolkit` is able to use MySQL providing useful informations like the number of trials, the current state, the gained reward or whatever info. The idea is quite simple: in section C you've run a script that has generated the database `MySqlLog` in MySQL and has created a new user called `prlt` with password `prlt`. Running an experiment `PolimiRLToolkit` loaded and used it, so that you can read datas about each trial and step, since your experiment is running. Main data are stored in table `AgentLog`.

As you can see, this table has a field named `ExpID`. It is used to let you to recognize the data of any single experiment: `PolimiRLToolkit` set it to 0 for the current experiment. You've to modify it by hand before starting a new experiment, so that every experiment will have a different ExpID and you will use it like a key.

In order to read them we provide a minimal MySQL tutorial:

- enter MySQL environment:

  ```
  mysql --password=prlt -u prlt MySqlLog
  ```

- list all databases:

  ```
  SHOW DATABASES;
  ```

- select `PolimiRLToolkit` database:

  ```
  USE MySqlLog;
  ```

- list all tables of the database:

  ```
  show tables;
  ```

- show the header of the table AgentLog:

  ```
  describe AgentLog;
  ```

- change `ExpID` from 0 to, eg, 15:

  ```
  update AgentLog set ExpID=15 where ExpID=0;
  ```

- delete the last experiment done:

  ```
  delete from AgentLog where ExpID=0;
  ```

- list all informations about the last experiment done:

  ```
  select * from AgentLog where ExpID=0;
  ```

you can read them according to this schema:

...	S	...
a	S^1	$R_a^{SS^1}$

where S is a state, a is the action done from that state and that will bring us to the state S^1, gaining he reward $R_a^{SS^1}$.

- like before, but reading data starting from Trial 201:

```
select * from AgentLog where ExpID=0 and Trial>200;
```

- have the number of trial done in experiment 21:

```
select max(Trial) from AgentLog where ExpID=21;
```

- sort data into a required order

```
select * from AgentLog where ExpId=0 order by trial,
   step;
```

As last example here we have the query you have to execute to write the average reward per trial of the experiment 10, to the ready to plot file `./tmp/avg.log`:

```
select Trial, avg(Reward) into outfile '/tmp/avg.log'
   from AgentLog where ExpID=10 group by Trial;
```

and to write the number of step per trial:

```
select Trial, count(*) as 'StepPerTrial' into outfile '/
   tmp/avg.log' from AgentLog where ExpID=10 group by
   Trial;
```

If your environment has a big number of state variables, remember that the default size of the column `CurrentState` is 20, so you've to enlarge it in order to log every state modification:

```
alter table AgentLog modify CurrentState varchar(50);
```

You can retrive more informations on:

- MySQL documentation in your computer:

  `/usr/share/doc/MySQL-common-<version>/manual.html#` *Tutorial*

- a SQL tutorial:
  `http://www.1keydata.com/sql/sql.html`

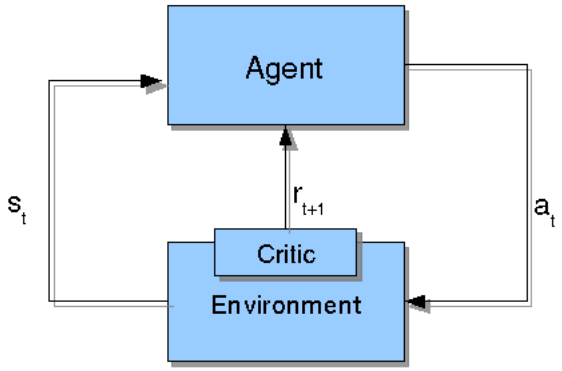


Figure 1: An agent interacting with an environment and rewarded by a critic

- another tutorial in more pages:
  `http://dev.mysql.com/doc/refman/5.1/en/tutorial.html`
- and another one in italian:
  `http://www.risorse.net/mysql/`

4 Understanding PRLT

4.1 Architecture overiew

Following the reinforcement learning paradigm, `PolimiRLToolkit` architecture is formed by an agent acting in an environment throught actions a, the perception of the state s and a reward r given by a critic like in Figure 1.

In order to achieve this result, `PolimiRLToolkit` has been divided in every aspect: we have a hierarchy of class representing various types of agents, environments, critics and algorithms. These are supported by classes representing variables (both state and action variables), containers of variables, q-tables and so on.

One of the basic building block of `PolimiRLToolkit` is the variable idea: to represent states and actions, we need variables. The toolkit provides a hierarchy of way to describe a variable, all deriving from `Variable` like in the diagram of Figure 2. Each object of this type is identified by a name and can be bounded or unbounded. Discrete variables inheredit from `DiscreteVariable`.

If the values of a `DiscreteVariable` have all the same distance one from another, then we have a `UniformDiscreteVariable`. An example is a variable whose domain are all natural numbers from 15 to 25. If the distance is not so regular, then you have to use a `GeneralDiscreteVariable` or to create your own class.

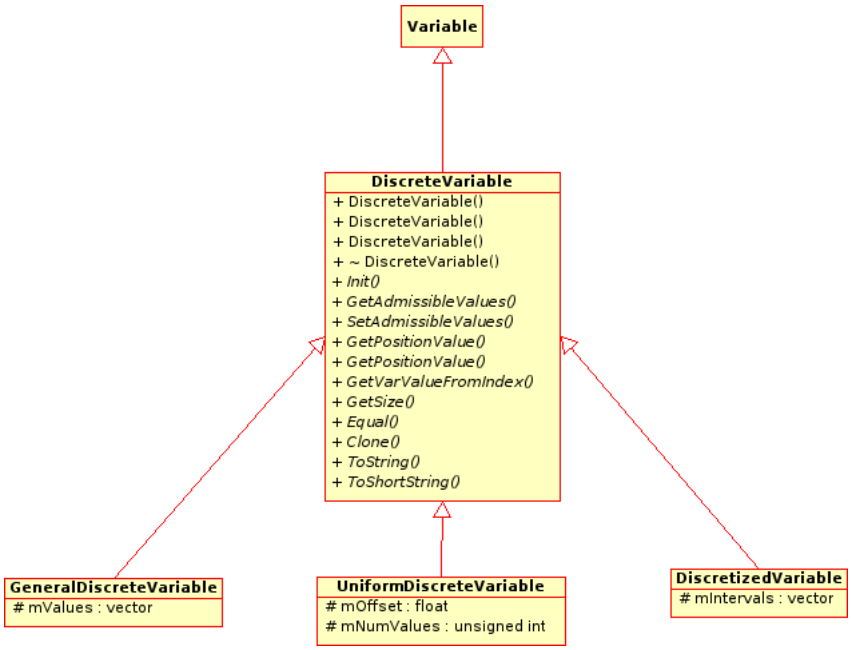


Figure 2: The hierarchy of variables

`DiscretizedVariable` is a discrete variable divided into intervals (see Figure 3).

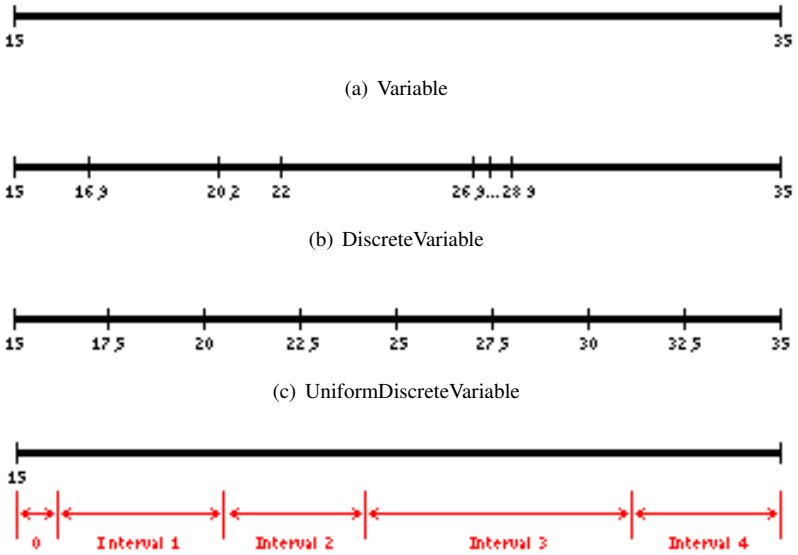
To manage variables the container `VariablesInfo` is provided. This container exposes methods to retrive a variable by its name or by its position in the collector. This collector class is used to exchange information between agents, environments and critics.

The class `Environment` is the parent of a family of classes representing the environment, its states and its dynamics. Each step of simulation is performed by the method `ExecuteActions` that receives the action the agent wants to execute as input, and returns the current state of the environment it represents, as the agent modified it (`GetStateDescription`). Other methods are `Reset()` and `ResetRandom()`. The first reset all state variables, so that the state returns to its initial configuration. The latter brings the environment in a valid random configuration.

If you have to develop new environments, all you have to do is to inherit from `Environment`, describe its dynamics and then create the appropriate description according to the `XSD` (see section 5). Thanks to this structure, you can interface a based on the reinforcement learning paradigm AI to any external application[5].

The description of an agent is more complicated. A RL agent is divided in two modules:

[5]If the environment you've created needs to interact with a java counterpart, you can use the JavaSocketHandler library.



(a) Variable

(b) DiscreteVariable

(c) UniformDiscreteVariable

(d) DiscretizedVariable

Figure 3: The same variable seen as a general continue variable with values from 15 to 35 (Figure 3(a)), seen as a DiscreteVariable (Figure 3(b)), as an UniformDiscreteVariable (Figure 3(c)), and as a DiscretizedVariable (Figure 3(d)),.

- the mapper, that transforms a multidimensional state description into a monodimensional one (the feature space). The base core class that implements this feature is `InputMapper`, whose main method is `GetFeature`.
- the agent, represented by the class `LearningInterface` (*not* by the class LearningAgent that is the "memory" of the agent itself!).

The method `Step` of `LearningInterface` will make the agent choose an action throught the method `Step` of `LearningAlgorithm`. In this method the agent has to store its new knowledge in some kind of structure, that can be a qtable, a func-

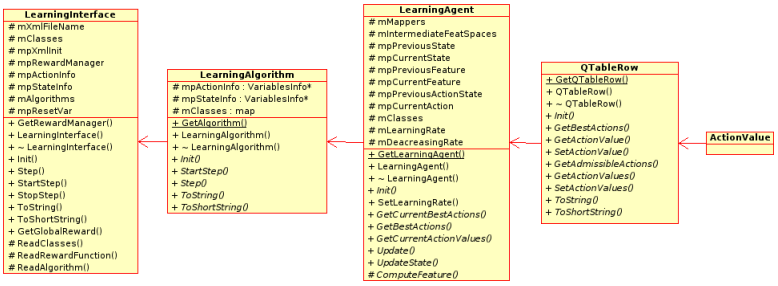


Figure 4: The set of classes representing an agent

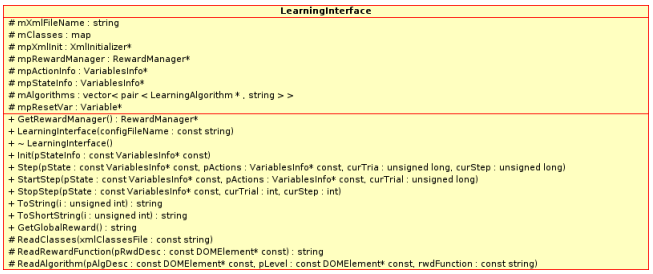


Figure 5: The LearningInterface class

tion approximator, or whatever else. This kind of structure is implemented by the class `LearningAgent`.

This contains a representation of the knowledge of the agent, that can be updated by the method `Update` and read using `GetCurrentBestActions` and other similar methods.

As an example of `LearningAgent`, we have the `TabularLearningAgent` that represents a q-table: this is one of the simplest way to store knowledge for a reinforcement learning agent. In `PolimiRLToolkit` it is represented by a collection of `QTableRow` objects, that are sets of `ActionValue` objects. The collection itself is representad by the `TabularLearningAgent`. In this way we have a container of all action values in each state of the environment.

4.2 Directory tree

Each one of the classes named in section 4.1 is inheredited in various way. To organize them, `PolimiRLToolkit` proposes its directory structure in the following way: all the project is contained in the `prlt` directory. This contains the main Makefile, some installation instructions and three main directories: `core`, `interfaces`, `shared`. and `utils`.

Direcorty `shared` contains the general Makefile and instructions to make the documentation.

`interfaces` contains libraries to connect `PolimiRLToolkit` to others applications:

- `JavaSocketHandler` is used to connect to java environments. It is a socket based communicator, so should work remotely.
- `NIPSBenchmark` contains the interface to create some test environment and a `PolimiRLToolkit` based agent. Throught it you can also test an instance of `PolimiRLToolkit` on the NIPS benchmark[6] defined by University of Al-

[6] `http://www.cs.rutgers.edu/~mlittman/topics/nips05-mdp/`

berta.

The most important directory in `prlt` is `core` that is divided in several directories that contain the description of the core classes of `PolimiRLToolkit` , and of some support classes. For *core classes* we define all classes that describe the agent, the environments, and the critic. Core classes have a hierarchical hierarchy: from the generic *base core classes* we create the *derived core classes*. As an example, from the base core class `Variable`, we have the derived core class `DiscretizedVariable`. For each core class we have a library that allows dynamic loading of derived classes. In this way the toolkit optimizes its memory usage by loading only useful classes. Since core classes are stored in precompiled libraries, we can get an istance of one of them by dynamically creating it.

Other subdirectories are:

- `prlt/core/bin`. It will contain the executable files of the project. This issue is not implemented, yet, so this directory is only a placeholder.

- `prlt/core/config` contains scripts to configure `PolimiRLToolkit` . The only instructions in it are currently those to set the MySQL logger and are only used in the configuration and installation phase.

- `prlt/core/data`: it contains classes.xml, that is a description of some classes used in the dynamic creation of each object. More in the detail, it contains three informations for each core class: `Class` is the name used in the XML descriptions of the esperiment. `Name` is the name of the method of that class that can return a new object of that type, `lib` is the path and the name of the library that contains the compiled code for that creation. In this way if you want to create a new class and you want an object of that type to be dynamically created, you must add in the file `Classes` this three informations. To create a new object deriving from a class in your c++ code, you must use the `Name` attribute you find in `Classes`. To create a reference to an object in the XML experiment description you will use the `Class` attribute. Other files in this directory have the same function, but less general. `prlt/core/data` contains two more directories: the first is `experiment`, where you will find the descriptions of the experiment done, grouped by algorithm. The second is `XSDTypes`, describing the grammar of each element of the `PolimiRLToolkit` experiment: the agent, the environment, the critic. If you will create a new algorithm, a new mapper, a new environment, or anything else that will be described by an XML in `PolimiRLToolkit` , then you will have to add its specification and XML grammar in this directory. Using XSD ensures `PolimiRLToolkit`

modularization and the creation of hierarchies describing the grammar of each experiment. See section 5 for more informations about this.

- `prlt/core/doc` here you will find useful documentations about the toolkit. Here will be compiled all doxygen documentation, too.
- `prlt/core/include` contains the base core classes's headers.
- `prlt/core/lib` is the repository of the libs used for the dynamic creation of core classes's instances. To each base core classes correspond a different file.
- `prlt/core/src` here we have all the sources of both core and support classes. Core base classes and support classes are just in this directory. Derived core classes are contained in a series of directories depending from the base core class they exetends.
  This directory also contains the main Makefile: the one in `prlt/` only calls this. Using this Makefile you can make all the project (`make` target), make the files in the test directory (`make tests`), or clean it (`make clean` or `make cleanall`).
- `prlt/core/test` is the repository of all the source code containing a main function, so here you find every c++ files that can generate an executable. This directory also stores all thier correspondent executables. We put in evidence `toolkit` used to run an experiment, and `tester` used to emulate an agent by hand in order to test an environment.

5 XML, XSD and experiments descriptions

Each experiment you can do using `PolimiRLToolkit` is described by an XML file[7] , so that you can change its parameters and test new algorithms and environment without recompile all the toolkit.
In `prlt/core/data/experiments` you will find some of this description files. These are wrote in XML and validated via XSD. Here we have a short XML example:

```
<element1/>
<element2/>
    <!--
        This is a comment.
    -->
```

[7] see prlt/core/data/experiment/qlearning/GridWorld.xml for a simple example. It describes an experiment with an agent learning on a gridworld using qlearning

```
<element3 attibute1=".." attibute2=".." attibute3="..">
    <element4 ...>
    <element4/>
    <element5 ...>
    ...
    <element5/>
<element1/>
```

In of `PolimiRLToolkit` all information is enclosed in the `Experiment` element. Its attributes and the attributes of the `XmlClasses` elements often are the same and are used to validate the document. All the others elements depend on the experiment you want to run, so that the general structure is like this:

```
<Experiment xmlns:xsd="http://www.w3.org/2001/XMLSchema-
   instance"
            xsd:noNamespaceSchemaLocation="../../XSDTypes
               /Prlt.xsd">
    <XmlClasses FileName="../data/Classes.xml"/>

    <ExperimentParameters>
        <Trials Num="1000"/>
        <MaxStepsPerTrial Num="500"/>
        <RandomSeed Seed="0"/>
        <RandomRestart Val="false"/>
    </ExperimentParameters>

    <EnvironmentDescription>
    <Data>
    <!-- environment paramenters -->
    </Data>
    <!-- Descripion of the state variable -->
    <StateVariable ...>
    <StateVariable ...>
    <StateVariable ...>
    ...
    </EnvironmentDescription>

    <AgentsDescription>
    <Agent>
        <AgentAlgorithm>
```

```
        ...
        </AgentAlgorithm>
        <Actuator>
        <!-- descriptions of what the agent can do -->
        ...
        </Actuator>
    </Agent>
    </AgentsDescription>

    <!-- Description of the critic -->
    <RewardManager>
    <RewardFunction>
    ...
    </RewardFunction>
    <RewardFunction>
    ...
    </RewardManager>
</Experiment>
```

As you can see the XML seems to reflect the `PolimiRLToolkit` paradigm of core classes: as a matter of fact in each section of it you can describe a base core classes as a complex structure based on derived core classes.

5.1 XML data parsing

First of all the XML description is read by the method `readClasses` of an object of type `Utils`[8] that uses xerces (see below) to validate the attributes of an element `Experiments` via XSD. The method `readClasses` is used by an object of the class `Experiment`. If some error occurrs then the program will exit signaling you what's wrong with the XML, otherwise the toolkit will continue.

At this point, the method `readClasses` divides the data read from the document into the logical sections corresponding to the `PolimiRLToolkit` architecture. These are usually elements of the XML, and are represented by the class `DOMElement` of xerces. The corrispondence between classes and XML elements is mapped in the file prlt/data/Classes.xml[9]

[8] Utils is a class containing some static attributes and methods for general usage

[9] so, if you write a new core or derived core class for the toolkit, remember to add its parameters to this file. Parameters are the class, the name of the method of that class that can return a new instance of it (it usually is declared as extern in the beginning of the class source code), the library containing the compiled code for the class (usually one *so.0.0 file of the directory in prlt/core/lib)

In this way every object of the toolkit gets only the reference to the DOMElement that describes it.

If you're writing classes for the toolkit, this is now the time to use `xerces`[10], an XML parser for c++. It can read and validate an XML document, and provides powerful methods to parse it. The method `Init` of every object that needs information from the XML, receives a `DOMElement` pointer, so that you can easily parse the XML fragment relative to your needs in this way:

- to read a list of element like

  ```
  <VariableQuantization Name="Row" Resolution="7" />
  <VariableQuantization Name="Col" Resolution="7" />
  ```

  use this commands:

  ```
  void MyClass::Init(const DOMElement* pDomState, ...)
  {
      // Create a list of the elements to read
      DOMNodeList* var_quant_elem_list = pDomState->
         getElementsByTagName(
          C2X(Utils::eVariableQuantization));
      // parse each element of the list
      for (unsigned int i = 0; i <var_quant_elem_list->
         getLength(); i++)
      {
          // var_quant_elem is the pointer to the
             single i-th element read:
          // if i=0 we will have the element name Row,
             if i=1 th element Col
          DOMElement* var_quant_elem = (DOMElement*)
             var_quant_elem_list->item(i);
          ...
          ...
      }
  }
  ```

- to read the attribute `name` of the element `StateVariable` in

  ```
  <StateVariable xsd:type="TUniqueDiscretizedVariable"
     Class="DiscretizedVar" Name="Pos">
  ```

[10] `http://xml.apache.org/xerces-c/index.html`

use this commands:

```
string var_name = X2C (var_quant_elem ->
   getAttributeNode(C2X(Utils::aName)) -> getValue())
    ;
```

As you can see, we do not directly use the strings `VariableQuantization` and `Name` in the c++ sections. As a matter of fact all elements and attribute names should be defined in the vocabulary file `prlt/core/src/Utils*`. These files define the class `Utils` cointaining a lot of useful, general porpouse static methods and attributes. In every class you will refer to each element and attribute found in a XML description throught a static attribute of the class `Utils`. In this way we can modify and have some centralized control of the vocabulary of `PolimiRLToolkit` XML definitions. We will usually refer to the XML element "foo" creating the attribute "eFoo" of the class `Utils`. In a parallel way we will usually refer to the XML attribute "foo" creating the attribute "aFoo" of the class `Utils`.

5.2 XSD in PRLT

XSD[11] stands for XML Schema Definition. It is also known as XML Schema Language, and represents an XML-based alternative to DTD. It describes the structure of an XML, but it also defines data types for elements and attributes, whether an element is empty or can include text, default and fixed values for elements and attributes, and so on.

`PolimiRLToolkit` uses it in order to validate XML experiments descriptions. The directory `prlt/core/data/XSDTypes` contains the set of .xsd files defining the grammar of every XML file. We have a main file and a file for each core class. The main file, `Prlt.xsd`, is the file included in the header of each XML experiment description:

```
<Experiment xmlns:xsd="http://www.w3.org/2001/XMLSchema-
   instance" xsd:noNamespaceSchemaLocation="../../
   XSDTypes/Prlt.xsd">
```

It contains all informations about the grammar of each experiment, depending by the class the experiment belongs to. `Prlt.xsd` starts including all other XSD files, which one describes every kind of core class. That same file ends reflecting the main structure of each experiment XML:

```
<xsd:complexType name="TExperiment">
```

[11] For more informations about XSD see `http://www.w3schools.com/schema/default.asp`

```
    <xsd:all>
        <xsd:element name="XmlClasses" type="TXmlClasses"
           />
        <xsd:element name="ExperimentParameters" type="
           TExperimentParameters"/>
        <xsd:element name="EnvironmentDescription" type="
           TEnvironmentDescription"/>
        <xsd:element name="AgentsDescription" type="
           TAgentsDescription"/>
        <xsd:element name="RewardManager" type="
           TRewardManager"/>
    </xsd:all>
</xsd:complexType>
```

and puts in evidence all main elements of the environment description.

Often have to specify the type of element you're describing[12] using attributes like `type="TExperimentParameters"` that signals to the parser that you're using an element of the XSD type `TExperimentParameters`

Thanks to this paradigm if you add some feature to the toolkit and need an XML validation for you feature, all you have to do is to add your grammar's description to the relative file in XSD directory.

[12] see the file GridWorld.xml

APPENDIX

A Savane CVS How to

`PolimiRLToolkit` is hosted in the web page `https://savane.elet.polimi.it/`. This site is "a central point for development, distribution and maintainance of software projects". It lets `PolimiRLToolkit` developers to have a their own community on the web, with a mailinglist, a cvs server and other services.

In order to partecipate to the development of the toolkit, you must register to savane and then subscribe to the project. This is the main way to use the cvs of `PolimiRLToolkit` project.

To register you must:

- open your browser at `http://savane.elet.polimi.it/`
- click on "Nuovo Utente via SSL"
- as "Nome Utente:" insert usr+yourSurname: eg, if your name is Mario Rossi, your "Nome Utente" will be `usrrossi`. You will use this name also in the second textbox.

  "Nome Reale:" is your real name.
- remember to insert you email. This will be used to notify the main events of the community, and will be used by others developers to contact you throught the savane mailing list. You can also use this mailing list to advise other users about changes you have done in the toolkit.

Now you just have to wait until the server admin will register you. This could take some hour. Once this will be done you can log from the main page by clicking on "Accedi via SSL" .

The next step is to generate a SSH Public and Private Keys. The command you must use from a console is:

```
ssh-keygen -t dsa
```

After logging on the server, click on "La Mia Configurazione", in the left. Then click "Edit the 1 SSH Public Key registered".

In "Chiavi autorizzate:", in the row "Chiave 1:", insert all the content of the file

```
~/.ssh/id_dsa.pub
```

This is the public key generated by the command above, and will be used to let the cvs server to identify you. If you will try to connect from another computer, you will need the same file in the same directory, so you have to copy it in the second computer (reusing ssh-keygen with the same passphrase should not be enough).

Is now the time to modify your ∼.bashrc to set some environments variables: open it and add the lines

```
export EDITOR=kwrite
export CVS_RSH=ssh
export CVSROOT=usrrossi@savane.elet.polimi.it:/cvs/prlt
```

The variable editor is optional. It is useful if you think to use the command line to interface to cvs. In this case every time you will commit your changes, the editor signaled by this variable will be opened letting you to keep a public diary of the modifications you've done to the code. In the example above `usrrossi` is to change with your username, as you've chosen creating the account on savane.

B Simple CVS How to

Here we have a list of useful commands if you don't think to use a GUI to interface to cvs.

- to download new code, or update the code you've on your hard disk you've to checkout:

  ```
  cvs co prlt
  ```

  the list of file checked will be shown: each to every file you will find one of this tags:

  - `U` in the case you've downloaded a new file. This happens if another developer created a new file and committed it.
  - `?` if you've created a new file and this is not in the common repository on the server.
  - `M` if the file you've on your computer is different from the one on the common repository. In this case cvs will save a copy of your file on your hard disk, and will try to merge your file.

- to upload a file that you've modified, but already exists on the repository, use

  ```
  cvs ci filename
  ```

  or, if you want to add a comment to the file,

  ```
  cvs ci -n "message" filename
  ```

- to add a new file to the repository

```
csv add filename
```

then you need a cvs commit for the same file. This operations have to be done in the same directory you've the files to upload.

- to delete a file from the common repository you've to remove it:

  ```
  cvs remove filename
  ```

  thereafter update the server via a cvs commit.

- to remove a file only from your computer, and not from the repository, it is good if you won't simply delete it, but use the command

  ```
  cvs release -d filename
  ```

  because the server could continue to think you've a file of the project in the position of the old one.

Remember that other developers are using the same repository.

Your code must compile!!

Others developers don't have to fix your bugs.

If you know other people is using your code, advise them about your modifications, and upload only tested code.

If you don't want to use the command line, you can use cvs graphical user interfaces such as tkcvs[13] or cervisia[14] that is a kde integrated gui. Another options, especially if you are already using it, is eclipse. In this last case you can read the appropriate instructions in section D

C Compiling MySQL++

Some distros (e.g. `Gentoo`) have such package ready to install. Other distros can download rpm, but this may not fit you distro's requirements, as you can see from MySQL++'s site, so could be useful compile MySQL++ from scratch, or compile MySQL, too.

Let start work:

1. download packages from `http://www.mysql.com/`

   MySQL -> the recommended version

   MySQL++ -> the last version

[13] `http://www.twobarleycorns.net/tkcvs.html`

[14] `http://cervisia.kde.org`

You can find it following the link Developer Zone -> Downloads, starting from the home page. At the moment they're at the page `http://dev.mysql.com/downloads/` and `http://tangentsoft.net/mysql++/`. If you want to start compiling MySQL sources than jump the next step, else go on.

2. install from RPM:
   if you've found MySQL on your distro's repository and want to use the rpm version of MySQL++, pay attention them to have the same version!

   ```
   rpm -Uvh MySQL-common-4.1.11-1mdk
   rpm -Uvh mysql++-2.0.6-1.rh9.i386.rpm
   rpm -Uvh libmysql14-devel
   ```

3. if you're compiling MySQL++ from scratch, then you could need MySQL-devel or libmysql14-devel (according to your version of mysql) in order to compile it. Then follow standard procedure:

   ```
   ./configure --prefix=/usr
   make
   make install
   ```

   using the /usr prefix could be useful in the path management.

C.0.1 MySQL++

We can now assume you've already installed MySQL.

Once you have done it you can start the MySQL server. This is generally done by setting up the mysql service. Some distros like SuSE, Mandriva and Ubuntu should have a GUI good for this. Anyway is mostly enough to type

```
/etc/init.d/mysql start
```

`PolimiRLToolkit` provides a script to generate the database that it will use. You can find the script in prlt/core/config, and you have to call it from that directory:

```
cd $(INST_DIR)/prlt/core/config/
./CreateDB.sh
```

Is now time to signal `PolimiRLToolkit` that you need it to write a log. This is done by modifiyng the file prlt/core/src/Makefile enabling the line

```
DEFS += -DLOGUP
```

simply removing number sign ♯ from that line [15]. Modifying this let us to recompile all `PolimiRLToolkit`:

```
cd $(INST_DIR)/prlt/core/src/
make cleanall
make tests
```

...have a coffee... and congratulations, you're ready to use the log like described in section 3.2.

D Eclipse

Eclipse is a development platform application framework you can use as an open source integrated development environment for various languages. You can downlaod it from the url `http://www.eclipse.org/downloads/`.[16].
Move the file downloaded[17] to the directory you want to install it, then unpack it:

```
tar xzf eclipse-SDK-3.1.2-linux-gtk.tar.gz
```

now you can run eclipse:

```
cd eclipse
./eclipse
```

and set you workspace: this is the default directory where eclipse will store every projects you may write.

Being `PolimiRLToolkit` mostly developed in c++, we will use the c++ plugin of eclipse: `cdt`. You can download it from the hotlink on the right of the same page where you've downloaded the sdk itself[18]. Follow the instructions you find on the site to get the Site Bookmark working: in the Help menu select "Software Updates" and then "Find and Install", then select "Search For New Features" to install and click "Next". Click "New Remote Site" to add an update with the URL provided on the site[19], and then click on "Finish". Select the new feature added in the textbox (I've simply called it "cdt"), press "Next", and go on following the instructions. It will download the plugin and install it.

Once you've installed cdt, in order to use eclipse as a cvs manager and to utilize all its features, you have to follow this procedure:

[15] Hint: you will also find useful to enable the line
DEFS += -DTOSTRING
in the same way, so that `PolimiRLToolkit` could more verbose

[16] Here we will refer to the last version, that at the moment is 3.1.2

[17] it should have a name like eclipse-SDK-3.1.2-linux-gtk.tar.gz

[18] `http://www.eclipse.org/downloads/`

[19] at the moment is `http://download.eclipse.org/tools/cdt/releases/eclipse3.1`

- create a new standard managed cpp project:
  on the eclipse menu choose File-> New-> Project, expand C++, then choose a "Standard Make C++ Project", then press "Next". Open the tag "C/C++ indexer". If you have more than 256MB of ram, choose Full c++ indexer, else use the CTags indexer (in some distros the executable is not `ctags` but `exuberant-ctags` or something similar). This last option is for code autocompletion. Having a full indexer to work will require about ten minutes for your computer to build the index, so this option will be modified once the full index will be complete and will be replaced by the faster CTags indexer (you need ctags to be installed). You will do this modifying from the option "Properties" of the menu "Project" of eclipse.

- import the toolkit from cvs:
  from the eclipse menu click on file -> Import, choose "Checkout Projects From CVS". As host insert `savane.elet.polimi.it`, and as "Repository path" insert `/cvs/prlt`. The user is the one you've created on savane. Leave the default connection type `extssh`. Click "Next".
  In the box labelled "Use specified module name" insert `prlt`, click "Next" and check out the project into an existing project. Select the project you've create two minutes ago ("rltoolkit" in my case). Click on "Finish".
  Now you have to wait until the indexer finish its work. It could be about ten minutes: you will see a status bar on the bottom of the eclipse main window.

- set the eclipse environment: open the C/C++ perspective: on the right you shuld find the "Make Targets" window. Open it, expand the tree until you can right click on the directory src. Select "Add Make Target" and create a make target named "tests" with make target "tests".

  Starting from this moment you can compile the toolkit by double clicking the appropriate target in the make target window.

You can also use eclipse to write the xml descriptions of an environemnt: what you have to do is to install the Web Tools Platform plugins.
Open your browser at the page `http://www.eclipse.org/webtools/`, click on "downloads" and on "installed via Update Manager", then follow the instruction[20].

[20] if your are expert and just want the Site Bookmark to add, you can use this one: `http://download.eclipse.org/webtools/updates/`

E Gnuplot essentials

Gnuplot[21] is "a portable command-line driven interactive data and function plotting utility for UNIX". You can use it to plot results of your experiments. In order to do this you need a file containing the data to plot in the format MySQL gives you (see section 3.2).
A file containing only one column of numbers will be interpreted as the array of data for y-axis assuming x-axis to have a linear growth. If the file contains two columns separated by a tab character, then each pair of data in a row will be read as coordinates of the point to draw.

Gnuplot provides you a shell environment you can enter simply typing `gnuplot` on your consolle. The main command you will use to paint a graph is

```
plot 'filename' with l
```

where `filename` is the name of the file containing the data to plot. Using `with l` will draw a line between each point of the graph, else a character + will be paint for each point descibed by the data file.

Gnuplot can also paint tridimensional graphs: all you have to do is to enter the Gnuplot shell, and give this commands:

```
set hidden3d
splot 'filename' w l palette
```

where the first command set the Gnuplot environment so that only visible faces of the graph will be painted. The second command plots the graph: `w l` is an shortcut for `with l`, palette use differents colors to put in evidence the heigh of each point in the graph.
Using this paint mode, you've also to wrote a different data file: this will contain three columns, the first for `x`, the second for `y` and the third for `z`. You will wrote a row for each point and an empty row between points with different `x`.

You can also paint contour lines or the grid:

```
set contour
set grid
```

To paint more than one file to the same plot use

```
plot 'filename1' with l, 'filename2' with l
```

You can also command gnuplot throught pipes. In this way you can easily write scripts able to plot datas:

[21] `http://www.gnuplot.info/`

```
echo "plot␣'filename'␣with␣l" | gnuplot -persist
```

If you have more needs you can use the `help` command of the gnuplot shell, or read this tutorial in the web: `http://t16web.lanl.gov/Kawano/gnuplot/index-e.html`

F Valgrind

Valgrind is a "suite of tools for debugging and profiling Linux programs." In the toolkit it is most used to:

- find memory leaks
- find incorrect freeing of memory
- find touched memory you shouldn't touch

To use it simply compile `PolimiRLToolkit` , then from a shell type the command:

```
cd prlt/core/test/
valgrind --tool=memcheck --leak-check=yes --log-file=<
   filename> ./toolkit parametri
```

where `<filename>` is the name of the file that will contain the log. The pid number of the process runned under valgrind will be added to `filename`.

If you want to find more errors try adding the options

```
--show-reachable=yes
```

that will let you to find memory areas still referenced by your program, also when it is terminated. With the command

```
valgind --tool=memcheck --leak-check=yes --log-file=<
   filename> --error-limit=no --num-callers=40 --leak-
   resolution=high --show-reachable=yes toolkit parametri
```

you will find every possible error and more.

To read more give a glance to `http://www.valgrind.org/` or to the quick start guide at `http://www.valgrind.org/docs/manual/quick-start.html`

G Callgrind and Kcachegrind

Callgrind (also known under the names Calltree and KCachegrind) is a profiling tool that builds on the Valgrind framework. Its graphical frontend is kcachegrind.

To use callgrind simply type

```
callgrind --log-file=logFile yourExecutable
```

where logFile is the file where the log will be wrote, while yourExecutable is the path to the executable to profile and the options to pass it.
Calling KCachegrind on the just created log file

```
kcachegrind logFile
```

you can dispone of a GUI showing you which methods of the executable you profiled has taken more resources.

For more informations see the following short how to and the kcachegrind home page:

`http://tools.openoffice.org/profiling/callgrind-howto.html`

`http://kcachegrind.sourceforge.net/cgi-bin/show.cgi`

H Useful links

`http://xml.apache.org/xerces-c/apiDocs/index.html`

`http://www.w3schools.com/`

`http://www.w3schools.com/xml/default.asp`

`http://www.w3schools.com/dtd/default.asp`

`http://www.w3schools.com/dom/default.asp`

`http://www.xmlfiles.com/`

`http://gcc.gnu.org/onlinedocs/libstdc++/latest-doxygen/`

I Acknowledgment

This document has been wrote thanks to:

. Andrea Bonarini - coordinator (bonarini@elet.polimi.it)

. Marcello Restelli - project manager (restelli@elet.polimi.it)

. Lazaric Alessandro - project manager (lazaric@elet.polimi.it)

. Lattuada Maurizio

. Mario Lozza

. Matteo Lazzarotto

. Simone Tognetti

. Vitali Patrick