Documentation for Python Extractor / Reflector

Catherine Canevet, Matthew Prowse

November 1, 2002

Contents

1 Running the Extractor **2**

2 Running the Reflector **3**

3 Overview of Modules **4**
- 3.1 Extractor.py . . . 4
- 3.2 PEPA_Extractor.py . . . 5
- 3.3 StateMachine.py . . . 5
- 3.4 Collaboration.py . . . 6
- 3.5 Cooperation.py . . . 6
- 3.6 Reflector.py . . . 6

4 Key Algorithms **7**
- 4.1 Generating the Terms . . . 7
- 4.2 Generating the System Equation . . . 9
- 4.3 Inserting into the Cooperation Tree . . . 11

5 Implementation Details **13**

A Example: Active Badge **14**

B Example: Server-Client **19**

C Program Listings **24**
- C.1 Extractor.py . . . 24
- C.2 PEPA_Extractor.py . . . 26
- C.3 Collaboration.py . . . 29
- C.4 StateMachine.py . . . 30
- C.5 Cooperation.py . . . 32
- C.6 Reflector.py . . . 35
- C.7 extract script . . . 37
- C.8 reflect script . . . 38

1 Running the Extractor

There is a Python script called *extract* which accepts parameters on the command line, and performs the extraction. The syntax is:

```
extract [-q|--quiet] [-n|--no-pepa] [-p|--pepa=FILE] [FILE]

-q, --quiet
    Do not print PEPA output to screen.
    Default is to print to screen.

-n, --no-pepa
    Do not write PEPA output to create a .pepa file.
    Default is to write to file.

-p=FILE, --pepa=FILE
    Write PEPA output to a given file.
    Default is input file with .pepa extension.

FILE
    File from which to extract.
    If omitted, the user will be prompted.
```

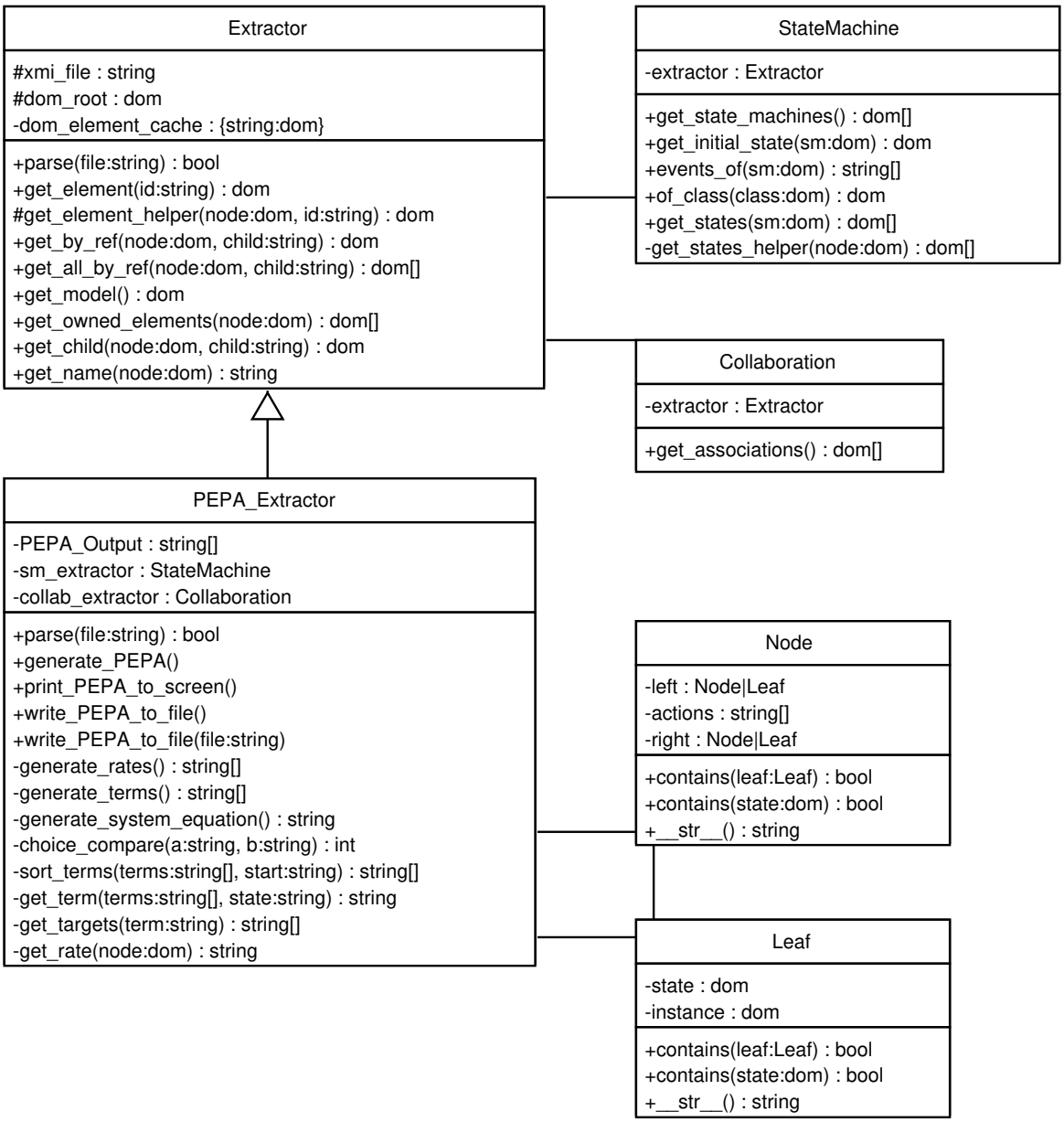
The extraction is performed by creating an object of class *PEPA_Extractor* and calling the *parse*() and *generate_PEPA*() functions, followed by *print_PEPA_to_screen*(), *write_PEPA_to_file*() or both.

2 Running the Reflector

There is another Python script called *reflect* which accepts parameters on the command line, and performs the reflection. The syntax is:

```
reflect [-c|--clear] [-n|--new=new_file] [xmi_file|zargo_file] [xml_file]

-c, --clear
    Remove reflected information the xmi_file or zargo_file.
    Any xml_file parameter will be ignored.

-n=new_file, --new=new_file
    Alternative file for reflected results.
    If omitted, the original input file will be altered.

xmi_file | zargo_file
    The original file for input to the reflector.
    If omitted, the user will be promted.

xml_file
    The xml file containing the PEPA workbench results.
    IF omitted, the user will be promted.
```

3 Overview of Modules



3.1 Extractor.py

The module *Extractor.py* contains one class called *Extractor*. An object of this class can perform a number of fundemental operations. The two key methods are *parse*() and *get_element*().

The *parse*() method takes a single argument, the name of the file to parse. Whether the file is an *.xmi* or a *.zargo* file, the *xml.dom.minidom* module parses this file, using its *parse*() and *parseString*() functions respectively. The method returns 1 for a successful parse, or 0 for an unsuccessful parse. A successful parse will result in two instance fields being initialised: *xmi_file* holding the input filename, and *dom_root*

holding a reference to the DOM model.

The *get_element*() method also takes a single argument, an *xmi.id* value, and returns the element in the DOM model whose *xmi.id* attribute matches this vale. If no such element is found, the method returns *None*.

A number of other methods and functions are provided in this module, performing useful operations on the DOM model or its elements. These include *get_child*(), returning the named child of a given node; *get_name*() returning the text value of the '.name' child of a given node; and *get_by_ref*(), returning the elements pointed to by the *xmi.idref* attributes of the children of the named child of a given node.

To improve the efficiency of the above methods, a dictionary (or hash table) called *dom_elemnt_cache* stores references to DOM elements indexed by the value of their *xmi.id* value. Elements are added to this cache by a helper method for *get_element*().

3.2 PEPA_Extractor.py

The *PEPA_Extractor* class defined in this module is a sub-class of the *Extractor* class. All methods of the *Extractor* class are inherited.

The primary method of this class is *generate_PEPA*(). It calls three further methods (*generate_rates*(), *generate_terms*() and *generate_system_equation*()) to fill an array *PEPA_output* with the PEPA syntax corresponding to the current model.

Two methods, *print_PEPA_to_screen*() and *write_PEPA_to_file*() provide access to the results. The latter takes an optional argument, the filename to which the results should be written. If omitted, the filename will be calcuated from the input file, *xmi_file*, and given a *.pepa* file extension.

A number of functions are present in this module, used to sort the behaviours within a term, and terms within a component. These produce more readable PEPA.

The *PEPA_Extractor* makes use of two further classes, *StateMachine* and *Collaboration*. The fields *sm_extractor* and *collab_extractor* contain references to instances of these classes, created by giving the constructor a reference to the *PEPA_Extractor* object itself.

3.3 StateMachine.py

This module contains a class called *StateMachine*. The constructor accepts an object of class *Extractor* which is later used to retreive elements from the DOM model.

The class and module provide methods and functions to extract information related to statemachines and their components from the model. These include *get_state_machines*(), returning an array of 'StateMachine' elements; *get_states*(), returning an array of 'State' elements from a given statemachine; and *events_of*(), returning an array of events present in a given statemachine.

This module acts as a filter, or a plugin, providing State Diagram specific extraction methods to any *Extractor* that instantiates it.

3.4 Collaboration.py

This module contains a class called *Collaboration*. The constructor accepts an object of class *Extractor* which is later used to retreive elements from the DOM model.

The class provides a method *get_associations*(), returning an array containing pairs of associated instances ('ClassifierRole' elements).

3.5 Cooperation.py

The *Cooperation.py* module contains the definitions for two classes: *Node* and *Leaf*. When the *PEPA_Extractor* is generating the system equation, it builds a tree consisting of *Node*s and *Leaf*s.

A *Node* represents a cooperation ($\bowtie$), consisting of left and right branches, and a set of synchronisers. A *Leaf* contains a state, and the class instance from the collaboration diagram that it represents. A *__str__*() method in both classes produces a string representation in PEPA syntax.

The recursive *contains*() method of the *Node* and *Leaf* classes take either a *Leaf* object or a *state* element as argument. If an object of class *Leaf* is given, the method returns 1 iff the an EXACT match is found (state and instance). If a *state* element is given, the method returns 1 iff another leaf containing the same state is found, regardless of the instance it is related to.

The module contains a function called *insert*() which takes two arguments of either *Node* or *Leaf* and inserts one into the other, returning the new *Node*.

3.6 Reflector.py

The *Reflector.py* module contains a class called *Reflector*. The class's *parse*() method performs as the *parse*() method of the *Extractor* class, taking the filename of either an *.xmi* or a *.zargo* file, and parsing the file with the *xml.dom.minidom* module.

The *reflect*() method takes an *.xml* file as an argument, parsing the file using the *xml.dom.minidom* module to extract the PEPA workbench results. The DOM model obtained by parsing the original *.xmi* file then has it's state names appended with the information from the *.xml* file.

The *clean*() method does not take any argument, and modifies the model obtained by parsing the original *.xmi* file by removing all reflected information from the state names.

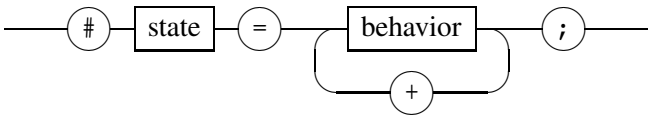
The *effect_changes*() method takes an optional parameter, a destination filename. An *.xmi* file is always valid for this parameter, although a *.zargo* file is only valid iff the original file was also a *.zargo*. If omitted, the original *.xmi* or *.zargo* file is updated with the reflections (or cleaned).

4 Key Algorithms

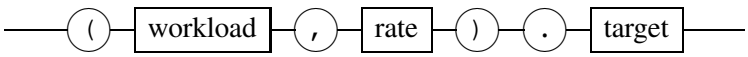
4.1 Generating the Terms

A *term* represents the behaviours that can exhibited by a component in a given state. The syntax for a *term* and each *behaviour* is shown below:

term



behaviour



Each *behaviour* consists of three parts: the workload, rate and target state. These correspond to the event name, action expression and target state of a transition respectively. Given a model, the following algorithm will compute all terms required for the PEPA output:

compute_terms( )

```
 1: terms ← empty list
 2: for each statemachine S do
 3:    for each state s (of S) do
 4:       behaviors ← empty list
 5:       for each outgoing transition t (of s) do
 6:          w ← name of trigger event of t
 7:          r ← contents of "rate(...)" expression of t
 8:          tgt ← name of target state of t
 9:          behaviors ← behaviors + "(w, r).tgt"
10:       end for
11:       n ← name of state s
12:       terms ← terms + "# n = behaviors_0 [ + behaviors_1 [ + ... ] ]"
13:    end for
14: end for
15: return terms
```

Line 1 declares an empty array called *terms*. As each term is generated, it is added to this array.

Lines 2-14 comprise a *for* loop to consider each statemachine in the model. Each statemachine should admit zero or more terms.

Line 3-13 comprise a *for* loop to consider each state within the current statemachine. Each state should correspond to exactly one term.

Line 4 declares an empty array called *behaviours*. As each behaviour is generated, it is added to this array.

Lines 5-10 comprise a *for* loop to consider each outgoing transition from the current state.

Lines 6-9 generate the behaviour corresponding to the current transition. The workload, the rate and the name of the target state are extracted for this.

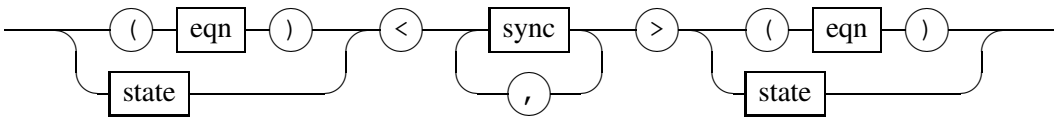
Lines 11-12 generate the term. The behaviours are separated by "+".

Line 15 returns the generated terms.

4.2 Generating the System Equation

The *system equation* represents the cooperations between components. It has the following syntax:

eqn



Given a model, the following algorithm will generate the system equation required for the PEPA output:

compute_system_equation( )

```
 1: cooperations ← empty list
 2: for each association a (of the collaboration diagram) do
 3:    left_instance ← left element of a
 4:    right_instance ← right element of a
 5:    left_sm ← statemachine of the class of left_instance
 6:    right_sm ← statemachine of the class of right_instance
 7:    left_leaf ← Leaf( initial state of left_sm, left_instance )
 8:    right_leaf ← Leaf( initial state of right_sm, right_instance )
 9:    sync ← ( events in left_sm ) ∩ ( events in right_sm )
10:    cooperations ← cooperations + Node( left_leaf, sync, right_leaf )
11: end for
12: top ← first element in cooperations
13: remove first element from cooperations
14: while cooperations is not empty do
15:    counter ← 0
16:    for each cooperation c (in cooperations) do
17:       top ← insert(top, c)
18:       if top has changed then
19:          remove c from cooperations
20:          counter ← counter + 1
21:       end if
22:    end for
23:    if counter is still 0 then
24:       stalemate has occurred - break from while loop
25:    end if
26: end while
27: sys_eqn ← convert top to string
28: return sys_eqn
```

Line 1 declares an empty array called *cooperations*. As each "atomic" cooperation is generated, it is added to this array.

Lines 2-11 comprise a *for* loops to consider each association in the collaboration diagram of the model. Each association should correspond to exactly one cooperation.

Lines 3-4 declare *left_instance* and *right_instance* to be the two participants in the association. (There is no significance as to which is left or right.)

Lines 5-6 declare *left_sm* and *right_sm* to be the statemachines belonging to the classes of which *left_instance* and *right_instance* are instances.

Lines 7-10 generate a cooperation: a *Node* consisting of two *Leaf* elements and a set of actions. This *Node* is added to the *cooperations* array.

Lines 12-13 remove the first cooperation (order is not important) from the *cooperations* array and call it *top*.

Lines 14-26 comprise a *while* loop that performs the body until the *cooperations* array is empty.

Lines 16-22 comprise a *for* loop that performs an iteration across the *cooperations* array and inserts each cooperation into *top*. If the insert is successful, the cooperation is removed from the *cooperations* array.

Lines 15, 20 and **23-25** declare and use a *counter* variable. It is possible for the insertion of a cooperation into *top* to fail, and the it will not be removed from the *cooperations* array. If an iteration across the *cooperations* array results in no successful inserts, the *while* loop is terminated.

Lines 27-28 return the string representation of the completed cooperation tree *top*.

4.3 Inserting into the Cooperation Tree

The following algorithm is used to combine two cooperation trees or parts thereof. The arguments *top* and *new* can either represent *Node* or *Leaf* elements of a cooperation tree. The *Node* that is returned is the result of inserting *new* into *top*:

insert(*top*, *new*, *times* = 0)

```
 1: if top is a Leaf and new is a Leaf then
 2:    return Node(top, [ ], new)
 3: else if top is a Leaf then
 4:    return new
 5: else if new is a Leaf then
 6:    if top.left contains another instance of new then
 7:       return Node( insert( top.left, new ), top.actions, top.right )
 8:    else if top.right contains another instance of new then
 9:       return Node( top.left, top.actions, insert( top.right, new ) )
10:    else
11:       return top unchanged
12:    end if
13: end if
14: if top.left contains new.left and top.right contains new.right then
15:    return Node( top.left, union( top.actions, new.actions ), top.right )
16: else if top.left contains new.left then
17:    if top.right contains another instance of new.right then
18:       if top.actions ∩ new.actions ≡ new.actions then
19:          return Node( top.left, top.actions, insert( top.right, new.right ) )
20:       end if
21:    end if
22:    return Node( insert( top.left, new ), top.actions, top.right )
23: else if top.right contains new.right then
24:    if top.left contains another instance of new.left then
25:       if top.actions ∩ new.actions ≡ new.actions then
26:          return Node( insert( top.left, new.left ), top.actions, top.right )
27:       end if
28:    end if
29:    return Node( top.left, top.actions, insert( top.right, new ) )
30: end if
31: if times ¿ 0 then
32:    return top unchanged
33: end if
34: swap left and right leaves of new
35: return insert( top, new )
```

Lines 1-13 comprise the "base cases". Either *top*, *new* or both *top* and *new* are *Leaf* elements.

Line 2 is performed if both *top* and *new* are *Leaf* elements. A new *Node* that is the parallel combination of the two leaves is returned.

Line 4 is performed if a *Node* (*new*) is being inserted into a *Leaf* (*top*). The *Node new* itself is returned.

Lines 6-12 recursively insert the *Leaf new* into either the right or left branch of *top* depending on the location of another instance of *new*.

Lines 14-30 comprise the cases where both *top* and *new* are *Node* elements.

Line 15 merges the synchronisers of the *top* and *new* cooperations and return the *Node top* with the (possible) additional synchronisers.

Line 22 recursively inserts *new* into the left branch of *top*.

Line 29 recursively inserts *top* into the right branch of *top*.

Lines 17-21 and **24-28** represent the possible need to form a parallel combination involving one of the leaves in *new*. The conditions in lines 21 and 30 ensure that such a parallel combination occurs only if the synchronisers already present maintain the requirements of the *new* cooperation. Otherwise, **line 22** or **29** will continue to insert the *Node new*.

Lines 31-33 ensure that an insertion is not tried indefinately. If a *times* argument is not given when calling the *isnert* function, the default value is 0.

Lines 34-35 reverse the *Node new* so that the left and right leaves are reversed, and try the insertion again with the *times* parameter equal to 1.

5 Implementation Details

In order to use the *xml.dom.minidom* package, at least Python 2.0 is needed. At least Python 2.2 is needed for static methods. Python 2.2 or greater is therefore needed to run the Extractor and Reflector.

The *insert*() function in the *Cooperation.py* module is implemented as a stand-alone function, not a method of any one class. If the *Extractor* were to be implemented in Java, the function would have to become a member methodof a class. At present, the function operates on two parameters, and returns a new (or the same) *Node*. It might make more sense to implement *insert*() as an instance method, effecting the changes to the *Node* on which it was called. The implementation of the given algorithm would then become fragmented and may be harder to follow.

A cache to speed up the response to repeated requests for DOM Elements with a given xmi.id value is used in the *Extractor* class, but does raise the issue of memory usage. The cache could potentially grow to be quite large. Disabling this cache could be provided as an option. The same technique is employed in the *StateMachine* class to speed up a number of methods.

The way in which the introduction of UML 2.0 will affect this implementation is unknown. As few details of the structure of the XMI as possible remain in the *PEPA_Extractor*, having been factored out to the *Extractor* class and individual 'diagram' extractor classes, such as *StateMachine*. Similarly, the effect of future versions of XMI is unknown.

The value of the *StateDiagram* and *Collaboration* modules as they are is uncertain. They are supposed to provide reusable code for Extractors that wish information to be extracted about the respective UML diagrams, but are pretty ad-hoc.

The current implementation uses a DOM parser, although there may need to be radical changes made if it were to be implemented using a SAX parser. The solution may involve a large number of 'cache' tables.

The software architecture has undergone a number of changes and has continued to evolve as the code has been refactored. The possible implementation in other programming languages has always been in mind. The techniques used here should be applicable beyond Python.

A Example: Active Badge

First of all, the output gives a list of the rates and their values. In our first section, we show how to we get the system terms (based on the state diagrams) , and in our second one the system equations (based on the collaboration diagram).

Generating the Terms

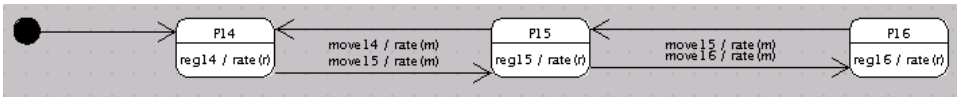


Figure 1: The Person state diagram

There are three states for this statemachine. The algorithm gives us :
P14 = (reg14, r).P14 + (move15, m).P15;
P15 = (move14, m).P14 + (reg15, r).P15 + (move16, m).P16;
P16 = (move15, m).P15 + (reg16, r).P16;

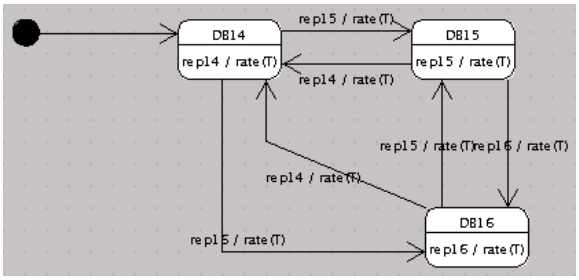


Figure 2: The Database state diagram

There are three states for this statemachine. The algorithm gives us :
DB14 = (rep14,infty).DB14 + (rep15,infty).DB15 + (rep16,infty).DB16;
DB15 = (rep14,infty).DB14 + (rep15,infty).DB15 + (rep16,infty).DB16;
DB16 = (rep14,infty).DB14 + (rep15,infty).DB15 + (rep16,infty).DB16;

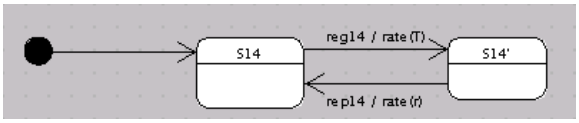


Figure 3: The Sensor S14 state diagram

S14 = (reg14, infty).S14';
S14' = (rep14, s).S14;

S15 = (reg15, infty).S15';
S15' = (rep15, s).S15;

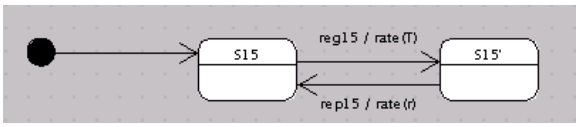


Figure 4: The Sensor S15 state diagram

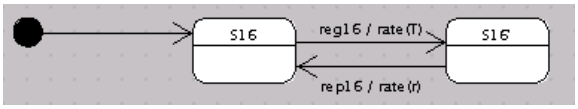


Figure 5: The Sensor S16 state diagram

S16 = (reg16, infty).S16';
S16' = (rep16, s).S16;

Generating the System Equation

Here is the collaboration diagram for this example :

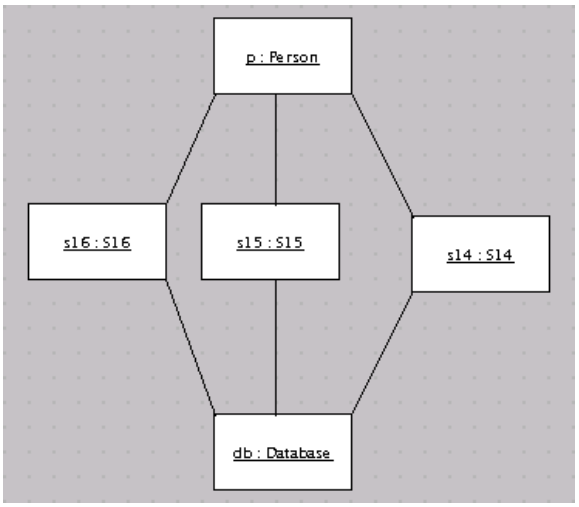


Figure 6: The collaboration diagram

There is no need for parallel combinations, so the *insert* algorithm used can be significantly reduced for clarity:

1: **if** top is a $Leaf$ **then**
2: **return** *new*
3: **end if**
4: **if** $top.left$ contains $new.left$ and $top.right$ contains $new.right$ **then**
5: take $top.actions$ as the union of $top.actions$ and $new.actions$

6: **return** top
7: **else if** $top.left$ contains $new.left$ **then**
8: insert *new* into $top.left$
9: **return** top
10: **else if** $top.right$ contains $new.right$ **then**
11: insert *new* into $top.right$
12: **return** top
13: **end if**
14: **if** $times == 1$ **then**
15: **return** top unchanged
16: **end if**
17: swap left and right leaves of *new*
18: insert *new* into top
19: **return** top

Consider the following (atomic) cooperations deduced from the collaboration diagram:

$$P14 \underset{reg15}{\bowtie} S15 \quad (1)$$

$$P14 \underset{reg16}{\bowtie} S16 \quad (2)$$

$$S15 \underset{rep15}{\bowtie} DB14 \quad (3)$$

$$S14 \underset{rep14}{\bowtie} DB14 \quad (4)$$

$$DB14 \underset{rep16}{\bowtie} S16 \quad (5)$$

$$P14 \underset{reg14}{\bowtie} S14 \quad (6)$$

Declare top

The first cooperation is removed from the list and is the start of our tree; referred to as top.

$$\mathbf{P14} \underset{reg15}{\bowtie} \mathbf{S15}$$

The list now consists of cooperations (2) to (6). An iteration across the list is performed as often as is necessary until all cooperations have been successfully inserted.

Try to insert (2)

$$\text{P14} \underset{reg16}{\bowtie} \text{S16}$$

Line 7 determines that the left leaf of (2) matches the left branch (leaf) of top (both P14), and so line 8 recursively calls insert() for lines 1-2 to return (2) as the new left branch of top:

$$\mathbf{(P14} \underset{reg16}{\bowtie} \mathbf{S16)} \underset{reg15}{\bowtie} \mathbf{S15}$$

The successful insertion removes (2) from the list, leaving cooperations (3) to (6).

Try to insert (3)

$$\text{S15} \underset{rep15}{\bowtie} \text{DB14}$$

None of the conditions in lines 1, 4, 7 or 10 are satisfied and so lines 17-19 retry the insertion, reversing (3):

$$\text{DB14} \underset{rep15}{\bowtie} \text{S15}$$

This time line 10 determines that the right leaf of the reversed (3) matches the right branch of top (both S15), and so line 11 recursively calls insert() for lines 1-2 to return the reversed (3) as the new right branch of top:

$$\textbf{(P14} \underset{reg16}{\bowtie} \textbf{S16)} \underset{reg15}{\bowtie} \textbf{(DB14} \underset{rep15}{\bowtie} \textbf{S15)}$$

The successful insertion removes (3) from the list, leaving cooperations (4) to (6).

Try to insert (4)

$$\text{S14} \underset{rep14}{\bowtie} \text{DB14}$$

The condition in line 10 is satisfied, so the function is called recursively on the right branch of top:

$$\text{DB14} \underset{rep15}{\bowtie} \text{S15}$$

None of the conditions in lines 1, 4, 7 or 10 are satisfied and so lines 17-19 retries the insertion, reversing (4):

$$\text{DB14} \underset{rep14}{\bowtie} \text{S14}$$

This time, the condition in line 7 is satisfied, so line 8 calls insert() recursively for lines 1-2 to return the reversed (4) as the new left branch of this right branch of top:

$$\textbf{(P14} \underset{reg16}{\bowtie} \textbf{S16)} \underset{reg15}{\bowtie} \textbf{((DB14} \underset{rep14}{\bowtie} \textbf{S14)} \underset{rep15}{\bowtie} \textbf{S15)}$$

The successful insertion removes (4) for the list, leaving cooperations (5) and (6).

Try to insert (5)

$$\text{DB14} \underset{rep16}{\bowtie} \text{S16}$$

None of the conditions in lines 1, 4, 7 or 10 are satisfied and so lines 17-19 retry the insertion, reversing (5):

$$\text{S16} \underset{rep16}{\bowtie} \text{DB14}$$

This time the condition in line 4 is satisfied, and lines 5-6 replace the set of actions on which the left and right branches of top synchronise by its union with the set of actions of (5):

$$\textbf{(P14} \underset{reg16}{\bowtie} \textbf{S16)} \underset{reg15,rep16}{\bowtie} \textbf{((DB14} \underset{rep14}{\bowtie} \textbf{S14)} \underset{rep15}{\bowtie} \textbf{S15)}$$

The successful insertion removes (5) from the list, leaving only (6).

Try to insert (6)

$$\mathrm{P14} \underset{reg14}{\bowtie} \mathrm{S14}$$

The condition in line 4 is satisfied, and lines 5-6 replace *top*'s set of actions by its union with the set of actions of (6):

$$\mathbf{(P14} \underset{reg16}{\bowtie} \mathbf{S16)} \underset{reg14,reg15,rep16}{\bowtie} \mathbf{((DB14} \underset{rep14}{\bowtie} \mathbf{S14)} \underset{rep15}{\bowtie} \mathbf{S15)}$$

The successful insertion removes (6) from the list, leaving no more (atomic) cooperations.

Convert to String

The resulting single tree can be converted to a string using an infix traversal to produce the system equation in PEPA syntax.

PEPA Output

Combining the terms and the system equation produces the corresponding PEPA model:

```
# P14  = (reg14, r).P14 + (move15, m).P15;
# P15  = (move14, m).P14 + (reg15, r).P15 + (move16, m).P16;
# P16  = (move15, m).P15 + (reg16, r).P16;

# DB14 = (rep14,infty).DB14 + (rep15,infty).DB15 + (rep16,infty).DB16;
# DB15 = (rep14,infty).DB14 + (rep15,infty).DB15 + (rep16,infty).DB16;
# DB16 = (rep14,infty).DB14 + (rep15,infty).DB15 + (rep16,infty).DB16;

# S14  = (reg14, infty).S14';
# S14' = (rep14, s).S14;

# S15  = (reg15, infty).S15';
# S15' = (rep15, s).S15;

# S16  = (reg16, infty).S16';
# S16' = (rep16, s).S16;

(P14 <reg16> S16) <reg14, reg15, rep16> ((DB14 <rep14> S14) <rep15> S15)
```

B Example: Server-Client

First of all, the output gives a list of the rates and their values. In our first section, we show how to we get the system terms (based on the state diagrams) , and in our second one the system equations (based on the collaboration diagram).

Generating the Terms

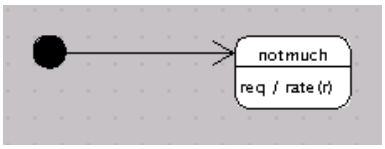


Figure 7: The User state diagram

There is only one state for this statemachine. The algorithm gives us :
notmuch = (q, r).notmuch

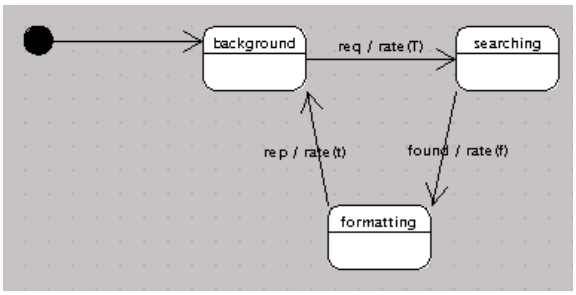


Figure 8: The Server state diagram

There are three states for this statemachine. The algorithm gives us :
background = (q, T).searching
searching = (found, f).formatting
formatting = (p, t).background

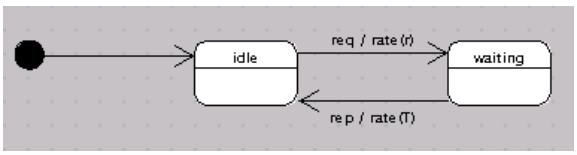


Figure 9: The Client state diagram

There are two states for this statemachine. The algorithm gives us :
idle = (q, r).waiting
waiting = (p, T).idle

Generating the System Equation

Here is the collaboration diagram for this example :

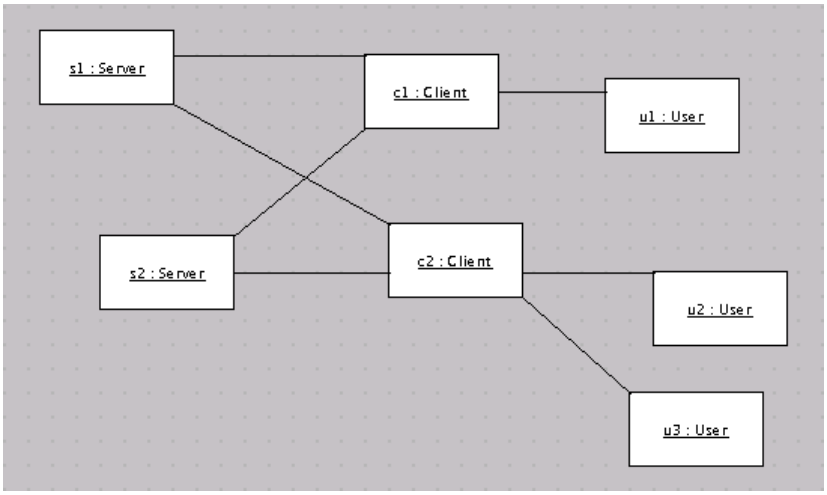


Figure 10: The collaboration diagram

The atomic cooperations are the following :

$$S1 \underset{req,rep}{\bowtie} C1 \tag{7}$$
$$S2 \underset{req,rep}{\bowtie} C2 \tag{8}$$
$$S1 \underset{req,rep}{\bowtie} C2 \tag{9}$$
$$U3 \underset{req}{\bowtie} C2 \tag{10}$$
$$U1 \underset{req}{\bowtie} C1 \tag{11}$$
$$S2 \underset{req,rep}{\bowtie} C1 \tag{12}$$
$$U2 \underset{req}{\bowtie} C2 \tag{13}$$

Step 1

The first cooperation is removed from the list and is the start of our tree; referred to as *top*.

$$\mathbf{S1} \underset{req,rep}{\bowtie} \mathbf{C1}$$

Step 2 : Try to insert (2)

top : S1 $\underset{req,rep}{\bowtie}$ C1
new : S2 $\underset{req,rep}{\bowtie}$ C2

It returns 'unable' so it twists the new one, but it still returns 'unable'. When it tries again, times equals 1, so it returns the top one, and we'll try to insert (2) again at the end.

Step 3 : Try to insert (3)

top : S1 $\underset{req,rep}{\bowtie}$ C1
new : S1 $\underset{req,rep}{\bowtie}$ C2
We have S1 in both left parts. We have C in both right parts. The intersection of the actions matches.
So it retrurns Node(top.left, top.actions, insert (top.right, new.right))

top : C1
new : C2
They are both instances so we have : C1 $\bowtie$ C2

And : **S1** $\underset{req,rep}{\bowtie}$ **(C1** $\bowtie$ **C2)**

Step 4 : Try to insert (4)

top : S1 $\underset{req,rep}{\bowtie}$ (C1 $\bowtie$ C2)
new : U3 $\underset{req}{\bowtie}$ C2
We have C2 in both right parts. But the left parts do not match. So it returns : Node(top.left, top.actions, insert(top.right, new))

top : C1 $\bowtie$ C2
new : U3 $\underset{req}{\bowtie}$ C2
C2 is in both right parts. The left parts do not match. So it returns Node(top.left, top.actions, insert(top.right, new)).

top : C2
new : U3 $\underset{req}{\bowtie}$ C2
Top is a leaf so it returns new.

And : **S1** $\underset{req,rep}{\bowtie}$ **(C1** $\bowtie$ **(U3** $\underset{req}{\bowtie}$ **C2))**

Step 5 : Try to insert (5)

top : S1 $\underset{req,rep}{\bowtie}$ (C1 $\bowtie$ (U3 $\underset{req}{\bowtie}$ C2))
new : U1 $\underset{req}{\bowtie}$ C1
We have C1 in both right parts. Left parts do not match. It returns Node (top.left, top.actions, insert(top.right, new))

top : C1 $\underset{}{\bowtie}$ (U3 $\underset{req}{\bowtie}$ C2)
new : U1 $\underset{req}{\bowtie}$ C1
No matches, it returns unable. It twists the new one which becomes : C1 $\underset{req}{\bowtie}$ U1
Then we have C1 in both left parts, and U in both right parts. But the intersection actions do not match. So it returns Node (insert(top.left, new), top.actions, top.right).

top : C1
new : C1 $\underset{req}{\bowtie}$ U1
Top is a leaf so it returns new.

And : **S1** $\underset{req,rep}{\bowtie}$ **((C1** $\underset{req}{\bowtie}$ **U1)** $\underset{}{\bowtie}$ **(U3** $\underset{req}{\bowtie}$ **C2))**

Step 6 : Try to insert (6)

top : S1 $\underset{req,rep}{\bowtie}$ ((C1 $\underset{req}{\bowtie}$ U1) $\underset{}{\bowtie}$ (U3 $\underset{req}{\bowtie}$ C2))
new : S2 $\underset{req,rep}{\bowtie}$ C1
C1 is in both right parts. S is in both left parts. The interaction of the actions matches. It returns Node(insert(top.left, new.left), top.actions, top.right).

top : S1
new : S2
They are both instances so we have : S1 $\underset{}{\bowtie}$ S2

And : **(S1** $\underset{}{\bowtie}$ **S2)** $\underset{req,rep}{\bowtie}$ **((C1** $\underset{req}{\bowtie}$ **U1)** $\underset{}{\bowtie}$ **(U3** $\underset{req}{\bowtie}$ **C2))**

Step 7 : Try to insert (7)

top : (S1 $\underset{}{\bowtie}$ S2) $\underset{req,rep}{\bowtie}$ ((C1 $\underset{req}{\bowtie}$ U1) $\underset{}{\bowtie}$ (U3 $\underset{req}{\bowtie}$ C2))
new : U2 $\underset{req}{\bowtie}$ C2
C2 is in both right parts but the left parts do not match. So it returns Node(top.left, top.actions, insert(top.right, new)).

top : (C1 $\underset{req}{\bowtie}$ U1) $\underset{}{\bowtie}$ (U3 $\underset{req}{\bowtie}$ C2)
new : U2 $\underset{req}{\bowtie}$ C2
C2 is in both right parts. U is in both left parts. The interaction of the actions does not match. So it returns Node(top.left, top.actions, insert(top.right, new)).

top : U3 $\underset{req}{\bowtie}$ C2
new : U2 $\underset{req}{\bowtie}$ C2
C2 is in both right parts, U is in both left parts and the interaction of the action matches. So it returns Node(insert(top.left, new.left), top.actions, top.right).

top : U3
new : U2
They are both instances so we have : U3 $\bowtie$ U2

$$\mathbf{(S1 \bowtie S2) \underset{req,rep}{\bowtie} ((C1 \underset{req}{\bowtie} U1) \bowtie ((U3 \bowtie U2) \underset{req}{\bowtie} C2)))}$$

Step 8 : Try to insert (2) again

top : (S1 $\bowtie$ S2) $\underset{req,rep}{\bowtie}$ ((C1 $\underset{req}{\bowtie}$ U1) $\bowtie$ ((U3 $\bowtie$ U2) $\underset{req}{\bowtie}$ C2)))
new : S2 $\underset{req}{\bowtie}$ C2
S2 is in both left parts and C2 is in both right parts so it returns Node (top.left, union(top.actions, new.actions), top.right), which is what we already had :

$$\mathbf{(S1 \bowtie S2) \underset{req,rep}{\bowtie} ((C1 \underset{req}{\bowtie} U1) \bowtie ((U3 \bowtie U2) \underset{req}{\bowtie} C2))}$$

PEPA Output

Combining the terms and the system equation produces the corresponding PEPA model:

```
# background = (q, T).searching
# searching = (found, f).formatting
# formatting = (p, t).background

# idle = (q, r).waiting
# waiting = (p, T).idle

# notmuch = (q, r).notmuch

(S1 <> S2) <req, rep> ( (C1 <req> U1) <> ( (U3 <> U2) <req> C2 ) )
```

C Program Listings

C.1 Extractor.py

```
# Extractor.py
# 24th July 2002
# Matthew Prowse

import xml.dom.minidom as dom  # to parse .xmi file
import zipfile  # to extract .xmi file from .zargo archive
import time


CONTENT             = "XMI.content"
MODEL               = "Model_Management.Model"
OWNED               = "Foundation.Core.Namespace.ownedElement"
NAME                = "Foundation.Core.ModelElement.name"


class Extractor:
    """A base class for extractors. Tailored for .xmi and .zargo files."""

    def __init__( self ):
        """Constructs a default Extractor."""

        # default instance fields for 'current' model
        self._xmi_file = ""
        self._dom_root = None
        self._dom_element_cache = {}



    def parse( self, file ):
        """uses xml.dom.minidom to parse a given file"""

        #<file> is an .xmi file
        if file.endswith( '.xmi' ):
            print "Parsing \""+file+"\""
            document_model = dom.parse( file )

        #<file> is a .zargo file
        elif file.endswith( '.zargo' ):
            z = zipfile.ZipFile( file )
            for f in z.namelist( ):
                if f.endswith( '.xmi' ):
                    file += "#" + f
                    print "Parsing \"" + file + "\""
                    document_model = dom.parseString( z.read( f ) )
                    break  # breaks the for loop

        # unknown file format
        else:
            print "Unable to parse \"" + file + "\": unknown format"
            return 0  # fail

        # set instance fields to reflect current model
        self._xmi_file = file
        self._dom_root = document_model.documentElement
        self._dom_element_cache = {}  # clear cached elements from any previous model

        self.get_element( "xmi.dummy" )  # fills cache unnaturally

        return 1  # succeed



    def get_model( self ):  # may return None
        """Returns the Model_Management.Model element."""
        content = get_child( self._dom_root, CONTENT )
        if content != None:
            model = get_child( content, MODEL )
            if model != None:
                return model



    def get_element( self, id ):  # may return None
        """Takes a string, and returns the DOM element with the matching xmi.id attribute."""
        if self._dom_root != None:
            obj = self._dom_element_cache.get( id, None )
            if obj == None:
                obj = self._get_element_helper( self._dom_root, id )
            return obj



    def _get_element_helper( self, node, string ):  # may return None
        """Helper function for finding DOM element with a given xmi.id attribute value."""
```

```
        for i in node.childNodes:
            if i.nodeType != 3 and i.hasAttribute( "xmi.id" ):
                id = i.getAttribute( "xmi.id" )
                self._dom_element_cache.setdefault( id, i )  # cache if found en-route
                if id == string: return i
            result = self._get_element_helper( i, string )
            if result != None: return result


    def get_by_ref( self, node, string ):  # may return None
        """returns the node pointed to by the only (first) child of the named child node"""
        all = self.get_all_by_ref( node, string )
        if len( all ) > 0:
            return all[0]


    def get_all_by_ref( self, node, string ):   # may return []
        """returns all the nodes pointed to by the children of the named child node"""
        result = []
        if self._dom_root != None:
            child = get_child( node, string )
            if child != None:
                for c in child.childNodes:
                    if c.nodeType != 3:  # ignore Text elements (eg. newline, tab, etc...)
                        idref = c.getAttribute("xmi.idref")
                        result.append( self.get_element( idref ) )
        return result



def get_owned_elements( node ):  # may return []
        result = []
        ownedElement = get_child( node, OWNED )
        if ownedElement != []:
                children = ownedElement.childNodes
                for c in children:
                        if c.nodeType != 3: result.append( c )
        return result


def get_child( node, string ):  # may return None
        """Returns the immediate child of a given node whose tag name ENDS with 'string' (or else None)"""
        if node != None:
                for child in node.childNodes:
                        if child.nodeType != 3:
                                if child.localName.endswith( string ):
                                        return child


def get_name( node ):  # may return ""
        """Returns the name of a given node."""
        name_node = get_child( node, NAME )
        if name_node != None:
                name = name_node.firstChild.nodeValue
                if name.find( '[' ) != -1:
                        name = name[ 0 : name.find( '[' )-1 ]
                return name
        return ""
```

C.2 PEPA_Extractor.py

```
import Extractor
import StateMachine
import Collaboration
from Cooperation import *


PSEUDOSTATE         = "Behavioral_Elements.State_Machines.Pseudostate"
OUTGOING            = "Behavioral_Elements.State_Machines.StateVertex.outgoing"
TARGET              = "Behavioral_Elements.State_Machines.Transition.target"
TRIGGER             = "Behavioral_Elements.State_Machines.Transition.trigger"
BASE                = "Behavioral_Elements.Collaborations.ClassifierRole.base"


class PEPA_Extractor( Extractor.Extractor ):

    def __init__( self ):
        """ Constructor calls Extractor constructor, and initialises PEPA_output array. """

        Extractor.Extractor.__init__( self )

        self._rates           = []  # needed by _get_rates() - filled by _get_terms()
        self._PEPA_output     = []  # cleared by parse() - filled by generate_PEPA()

        self.sm_extractor     = None
        self.collab_extractor = None



    def parse( self, file ):
        """ Parses a given file using the parse() method of Extractor (base class). """
        if file != "":
            self._PEPA_output = []  # clear old PEPA
            if Extractor.Extractor.parse( self, file ):  # call parse() in super class
                self.sm_extractor = StateMachine.StateMachine( self )
                self.collab_extractor = Collaboration.Collaboration( self )



    def generate_PEPA( self ):
        """ Initiates the extraction to PEPA, resulting in the _PEPA_output array. """
        terms = self._generate_terms( )
        rates = self._generate_rates( )
        syseqn = self._generate_system_equation( )

        self._PEPA_output = rates
        self._PEPA_output.extend( terms )
        self._PEPA_output.append( syseqn )



    def print_PEPA_to_screen( self ):
        """ prints PEPA output """
        for line in self._PEPA_output:
            print line



    def write_PEPA_to_file( self, file="" ):
        """ Creates a file containing the PEPA output. Will determine the filename if none given. """

        if file == "":
            # convert filename to .pepa
            file = self._xmi_file
            file = file[ 0 : file.rfind( "." ) ]  # strip trailing .xmi extension
            if file.rfind( "#" ) != -1:  # if a .zargo archive
                file = file[ file.rfind( "#" ) + 1 : len( file ) ]  # strip archive name
            file = file + ".pepa"  # add .pepa extension

        print "Writing PEPA output to \"" + file + "\""

        try:
            handle = open( file, "w" )
        except IOError, message:
            print >> sys.stderr, "File could not be opened:", message
            sys.exit( 1 )

        for line in self._PEPA_output:
            print >> handle, line

        handle.close( )



    def _generate_rates( self ):
        """ returns PEPA rate lines (or else []) """
        default_value = "2.0"
        rates = []
        for r in self._rates:
```

```
            rates.append( "% " + r + " = " + default_value + ";" )
        rates.sort()
        rates.append("")
        return rates


def _generate_terms( self ):
    """ returns PEPA terms (or else []) """
    terms = []  # list of all terms

    for sm in self.sm_extractor.get_state_machines( ):  # look at each statemachine

        sm_terms = []  # list of terms for THIS statemachine
        for s in StateMachine.get_states( sm ):  # look at each state in the statemachine
            if s.tagName == PSEUDOSTATE:
                continue

            outgoing = self.get_all_by_ref( s, OUTGOING )
            choices = []  # list of all behaviours for this state

            for t in outgoing:  # for every outgoing transition from the state
                target = self.get_by_ref( t, TARGET )
                trigger = self.get_by_ref( t, TRIGGER )
                rate = get_rate( t )

                if rate != "infty" and rate[0].isalpha():
                    if rate not in self._rates:  self._rates.append( rate )  # note rate

                choice = "(" + Extractor.get_name( trigger ) + "," + rate + ")." + Extractor.get_name( target )
                choices.append( choice )

            choices.sort( choice_compare )  # using a compare-function

            term = "# " + Extractor.get_name( s ) + "\t= " + " + ".join( choices ) +";"
            sm_terms.append( term )

        initial = self.sm_extractor.get_initial_state( sm )
        sm_terms = sort_terms( sm_terms, Extractor.get_name( initial ) )  # sort the terms
        terms.extend( sm_terms )  # add terms for THIS statemachine to all terms
        terms.append("")

    return terms


def _generate_system_equation( self ):
    """ returns PEPA system equation (or else \"\") """
    sys_eqn = ""
    cooperations = []  # list of all ('atomic') cooperations

    # create individual cooperation nodes representing associations between instances
    for (a_instance, b_instance) in self.collab_extractor.get_associations( ):

        # each association corresponds to a cooperation

        a_class  = self.get_by_ref( a_instance, BASE )
        b_class  = self.get_by_ref( b_instance, BASE )

        a_sm     = self.sm_extractor.of_class( a_class )
        a_events = self.sm_extractor.events_of( a_sm )

        b_sm     = self.sm_extractor.of_class( b_class )
        b_events = self.sm_extractor.events_of( b_sm )

        # build cooperation node
        a_leaf   = Leaf( self.sm_extractor.get_initial_state( a_sm ), a_instance )
        b_leaf   = Leaf( self.sm_extractor.get_initial_state( b_sm ), b_instance )
        #ab_node  = NodeX( a_leaf, intersection( a_events, b_events ), b_leaf )
        ab_node  = Node( a_leaf, intersection( a_events, b_events ), b_leaf )
        cooperations.append( ab_node )  # add cooperation node to list

    # combine all cooperations into one tree
    if len( cooperations ) > 0:
        top = cooperations[0]
        cooperations.remove(top)

        while len( cooperations ) > 0:
            counter = 0
            for c in cooperations:
                new_top = insert( top, c )
                if new_top != top:  # successful insert
                #if top.insert( c ) == 1:
                    cooperations.remove( c )
                    counter += 1
                    top = new_top
                else:
                    print "failure"

            if counter == 0:  # failure to insert ANY remaining cooperations
                print "stalemate"
```

```
                for c in cooperations:
                    print "Could not insert:\t",c.__str__()
                break  # break the while loop

        sys_eqn = top.__str__()  # format the tree as a string

    return sys_eqn


def choice_compare( a, b ):  # returns {-1|1}
    """ Compare function to sort behaviors lexicographically on target state. """
    if a[ a.rfind( "." ) : len( a ) ] < b[ b.rfind( "." ) : len( b ) ]:
            return -1
    return 1


def sort_terms( unsorted, start ):  # may return []
    """ Sorts a list of terms into the most logical ordering. """
    if start == None:  return unsorted  # no initial state

    initial = get_term( unsorted, start )
    if initial == None:  return unsorted  # failed to find term defining initial state
    sorted = [ initial ]
    unsorted.remove( initial )

    while len( unsorted ) > 0:
        counter = 0
        for s in sorted:
            targets = get_targets( s )
            targets.sort()  # just in case
            for target in targets:
                term = get_term( unsorted, target )
                if term != None:
                    sorted.append( term )
                    unsorted.remove( term )
                    counter += 1
        if counter == 0:
            print "stalemate"
            for u in unsorted:
                print "Could not sort \"",u,"\""
            break
    return sorted


def get_term( terms, state ):  # may return None
    """ From a list of terms: 'terms', returns the term that defines the behaviour of a given state: 'state'. """
    for l in terms:
            l_state = l[ l.find( "#" ) + 1 : l.find( "=" )-1 ].strip()
            if l_state == state:
                    return l


def get_targets( term ):  # may return []
    """ Returns array of the target states in a given term. """
    targets = []
    options = term[ term.find( "=" ) : term.rfind( ";" ) ].split( "+" )
    for o in options:
            # term = '(w,r).tgt'  - only want 'tgt'
            target = o[ o.rfind( "." )+1 : len( o ) ].strip()
            targets.append( target )
    return targets


def get_rate( node ):  # returns string
    """ Returns the rate of a given transition. """
    effect = Extractor.get_child( node, ".effect" )
    call_action = Extractor.get_child( effect, ".CallAction" )
    script = Extractor.get_child( call_action, ".script" )
    action_exp = Extractor.get_child( script, ".ActionExpression" )
    rate = Extractor.get_child( action_exp, ".body" ).firstChild.nodeValue
    if rate.startswith( "rate(" ):
            rate_string = rate[5:-1]
            if rate_string == "T": rate_string = "infty"
            return rate_string
    return "?"
```

C.3 Collaboration.py

```
import Extractor

class Collaboration:

    def __init__( self, extractor ):
        self.extractor = extractor  # class Extractor - although instance is PEPA_Extractor



    def get_associations( self ):  # may return []
        """ Returns a list of all AssociationRole elements in the model. """
        associations = []

        model = self.extractor.get_model( )
        if model != None:
            for a in Extractor.get_owned_elements( model ):
                if a.tagName.endswith( ".AssociationRole" ):
                    connection = Extractor.get_child( a, ".connection" )
                    if connection == None:  continue

                    end_roles = []
                    for c in connection.childNodes:
                        if c.nodeType != 3:  end_roles.append( c )
                        if len( end_roles ) != 2:  continue

                        assoc = ( self.extractor.get_by_ref( end_roles[0], ".type" ), self.extractor.get_by_ref( end_roles[1], ".type" ) )
                        associations.append( assoc )
        return associations
```

C.4 StateMachine.py

```
import Extractor


CLASS                = "Foundation.Core.Class"

STATEMACHINE         = "Behavioral_Elements.State_Machines.StateMachine"
CONTEXT              = "Behavioral_Elements.State_Machines.StateMachine.context"
TOP                  = "Behavioral_Elements.State_Machines.StateMachine.top"

PSEUDOSTATE          = "Behavioral_Elements.State_Machines.Pseudostate"
KIND                 = "Behavioral_Elements.State_Machines.Pseudostate.kind"

COMPOSITESTATE       = "Behavioral_Elements.State_Machines.CompositeState"
SUBVERTEX            = "Behavioral_Elements.State_Machines.CompositeState.subvertex"

OUTGOING             = "Behavioral_Elements.State_Machines.StateVertex.outgoing"
TARGET               = "Behavioral_Elements.State_Machines.Transition.target"
TRIGGER              = "Behavioral_Elements.State_Machines.Transition.trigger"


class StateMachine:

    def __init__( self, extractor ):
        self.extractor = extractor  # :Extractor

        self.statemachines_cache = []
        self.initial_state_cache = {}
        self.event_cache = {}
        self.class_cache = {}



    def get_state_machines( self ):
        """ Returns a list of all StateMachine elements defined in the model. """
        if self.statemachines_cache == []:
            model = self.extractor.get_model( )
            if model != None:
                for c in Extractor.get_owned_elements( model ):
                    if c.tagName == CLASS :
                        for o in Extractor.get_owned_elements( c ):
                            if o.tagName == STATEMACHINE:
                                self.statemachines_cache.append( o )
        return self.statemachines_cache



    def get_initial_state( self, sm ):  # may return None
        """ Returns the initial state of a given statemachine. """
        if self.initial_state_cache.get( sm, None ) == None:
            for s in get_states( sm ):
                if s.tagName == PSEUDOSTATE:
                    kind = Extractor.get_child( s, KIND )
                    if kind.getAttribute( "xmi.value" ) == "initial":
                        outgoing = self.extractor.get_by_ref( s, OUTGOING )
                        target = self.extractor.get_by_ref( outgoing, TARGET )
                        self.initial_state_cache[ sm ] = target
        return self.initial_state_cache.get( sm, None )



    def events_of( self, sm ):  # may return []
        """ Returns the events of a given statemachine. """
        if self.event_cache.get( sm, None ) == None:
            result = []
            for s in get_states( sm ):
                outgoing = self.extractor.get_all_by_ref( s, OUTGOING )
                for o in outgoing:
                    event = Extractor.get_name( self.extractor.get_by_ref( o, TRIGGER ) )
                    if event == "": continue
                    if event not in result:
                        result.append( event )
            self.event_cache[ sm ] = result
        return self.event_cache[ sm ]



    def of_class( self, clss ):  # may return None
        """ Returns the statemachine of a given class. (Every statemachine has a context.) """
        if self.class_cache.get( clss, None ) == None:
            for sm in self.get_state_machines( ):
                this_clss = self.extractor.get_by_ref( sm, CONTEXT )
                self.class_cache[ clss ] = sm
                if clss == this_clss: break
        return self.class_cache.get( clss, None )



def get_states( sm ):  # may return []
```

```
    """ Returns all states of a given StateMachine. """
    top = Extractor.get_child( sm, TOP )
    if top != None:
        cs = Extractor.get_child( top, COMPOSITESTATE )
        if cs != None:
            return _get_states_helper( cs )
        return []


def _get_states_helper( cs ):  # may return []
    """ Returns all states of a given CompositeState. (recursive) """
    sv = Extractor.get_child( cs, SUBVERTEX )
    result = []
    if sv != None:
        children = sv.childNodes
        for c in children:
            if c.nodeType != 3:
                result.append( c )
                if c.tagName == COMPOSITESTATE:
                    result.extend( _get_states_helper( c ) )
    return result
```

C.5 Cooperation.py

```
from Extractor import get_name

class Node:
    """ A class representing a cooperation. Branches can be of either Node or Leaf. """

    def __init__( self, left, actions, right ):
        """ Constructor creates Node corresponding to given arguments. """
        self.left = left
        self.actions = actions
        self.right = right

    def __str__( self ):
        """ Creates a string representation of the Node. """
        left = self.left.__str__()
        if isinstance( self.left, Node ):  left = "(" + left + ")"
        actions = "<" + ",".join( self.actions ) + ">"
        right = self.right.__str__()
        if isinstance( self.right, Node ):  right = "(" + right + ")"
        return left + actions + right

    def contains( self, leaf ):  # leaf:string|Leaf (exact match if Leaf)
        """ Returns true if 'self' contains a given leaf (or leaf of a given name) """
        return self.left.contains( leaf ) or self.right.contains( leaf )



def insert( top, new, times=0 ):
    """ Returns a new node that is a result of inserting 'new' into 'top', where 'new' and 'top' can be either of class Node or Leaf. """

    # base cases (either or both or top and new are leaves)
    if isinstance( top, Leaf ) and isinstance( new, Leaf ):
        return Node( top, [], new )

    elif isinstance( top, Leaf ):
        return new

    elif isinstance( new, Leaf ):
        if top.left.contains( new.state ):
            return Node( insert( top.left, new ), top.actions, top.right )
        elif top.right.contains( new.state ):
            return Node( top.left, top.actions, insert( top.right, new ) )
        return top


    # 'step' cases (both top and new are nodes)
    elif top.left.contains( new.left ) and top.right.contains( new.right ):
        return Node( top.left, union( top.actions, new.actions ), top.right )

    elif top.left.contains( new.left ):
        if top.right.contains( new.right.state ):  # possible need to parallelise
            if intersection( top.actions, new.actions ) == new.actions:
                return Node( top.left, top.actions, insert( top.right, new.right ) )
        return Node( insert( top.left, new ), top.actions, top.right )

    elif top.right.contains( new.right ):
        if top.left.contains( new.left.state ):  # possible need to parallelise
            if intersection( top.actions, new.actions ) == new.actions:
                return Node( insert( top.left, new.left ), top.actions, top.right )
        return Node( top.left, top.actions, insert( top.right, new ) )


    # unable to insert new into top
    if times == 1:  return top
    return insert( top, Node( new.right, new.actions, new.left ), times+1 )



class Leaf:
    """ A class to represent a leaf of a cooperation, ie. state. """

    def __init__( self, state, role ):
        """ Constructor to create Leaf from arguments. """
        self.state = state
        self.role = role

    def __str__( self ):
        """ Returns string representation of the leaf. """
        return get_name( self.state ) # + "(" + self.role + ")"

    def contains( self, leaf ):
        """ Returns true if 'self' matches a given leaf (or is at least the same state). """
        if isinstance( leaf, Leaf ):  # exact match required
            return ( leaf.state == self.state ) and ( leaf.role == self.role )
        return ( leaf == self.state )
```

```
def intersection( a_list, b_list ):  # may return []
        """ Returns the intersection of two lists, a_list and b_list. """
        intersection = []
        for a in a_list:
                if a in b_list:
                        intersection.append( a )
        return intersection



def union( a_list, b_list ):  # may return []
        """ Returns the union of two lists, a_list and b_list. """
        union = []
        for a in a_list:
                union.append( a )
        for b in b_list:
                if b not in union:
                        union.append( b )
        return union
```

```
class NodeX:
    """ A class representing a cooperation. Branches can be of either Node or Leaf. """

    def __init__( self, left, actions, right ):
        """ Constructor creates Node corresponding to given arguments. """
        self.left = Branch( left )
        self.actions = actions
        self.right = Branch( right )

    def __str__( self ):
        """ Creates a string representation of the Node. """
        left = self.left.__str__()
        #if isinstance( self.left, Node ):  left = "(" + left + ")"
        actions = "<" + ",".join( self.actions ) + ">"
        right = self.right.__str__()
        #if isinstance( self.right, Node ):  right = "(" + right + ")"
        return left + actions + right

    def contains( self, leaf ):  # leaf:string|Leaf (exact match if Leaf)
        """ Returns true if 'self' contains a given leaf (or leaf of a given name) """
        return self.left.contains( leaf ) or self.right.contains( leaf )



    def twist( self ):
        temp = self.left
        self.left = self.right
        self.right = temp



    def insert( self, new, times=0 ):
        """ Returns a new node that is a result of inserting 'new' into 'top', where 'new' and 'top' can be either of class Node or Leaf. """
        if isinstance( new, Leaf ):
            if self.left.contains( new.state ):
                return self.left.insert( new )
            elif self.right.contains( new.state ):
                return self.right.insert( new )

        # 'step' cases (both top and new are nodes)
        elif self.left.contains( new.left.tgt ) and self.right.contains( new.right.tgt ):
            self.actions = union( self.actions, new.actions )
```

```
            return 1

        elif self.left.contains( new.left.tgt ):
            if self.right.contains( new.right.tgt.state ):  # possible need to parallelise
                if intersection( self.actions, new.actions ) == new.actions:
                    return self.right.insert( new.right.tgt )
            return self.left.insert( new )

        elif self.right.contains( new.right.tgt ):
            if self.left.contains( new.left.tgt.state ):  # possible need to parallelise
                if intersection( self.actions, new.actions ) == new.actions:
                    return self.left.insert( new.left.tgt )
            return self.right.insert( new )


        # unable to insert new into top
        if times == 1:  return 0
        new.twist( )
        return self.insert( new, times+1 )



class Branch:
    def __init__( self, tgt ):
        self.tgt = tgt

    def __str__( self ):
        str = self.tgt.__str__()
        if isinstance( self.tgt, NodeX ):
            str = "(" + str + ")"
        return str

    def contains( self, leaf ):
        return self.tgt.contains( leaf )

    def insert( self, new ):
        if isinstance( self.tgt, NodeX ):
            return self.tgt.insert( new )
        else:
            if isinstance( new, NodeX ):
                self.tgt = new
                return 1
            else:
                self.tgt = NodeX( self.tgt, [], new )
                return 1
```

C.6 Reflector.py

```
import xml.dom.minidom as dom
from Extractor import get_name, get_child
import sys
import zipfile


class Reflector:
    def __init__( self ):
        self.xmi_file = ""
        self.xmi_doc = None
        self.xmi_root = None



    def parse( self, xmi_file ):

        #<file> is an .xmi file
        if xmi_file.endswith( '.xmi' ):
            print "Parsing \""+xmi_file+"\""
            self.xmi_doc = dom.parse( xmi_file )

        #<file> is a .zargo file
        elif xmi_file.endswith( '.zargo' ):
            z = zipfile.ZipFile( xmi_file )
            for f in z.namelist( ):
                if f.endswith( '.xmi' ):
                    print "Parsing \"" + xmi_file + "#" + f + "\""
                    self.xmi_doc = dom.parseString( z.read( f ) )
                    break  # breaks the for loop
            z.close( )

        # unknown file format
        else:
            print "Unable to parse \"" + file + "\": unknown format"
            return 0

        self.xmi_file = xmi_file
        self.xmi_root = self.xmi_doc.documentElement
        return 1



    def reflect( self, xml_file ):
        # No checks for duplicate state names
        # Should check have right component before reflecting.

        print "Reflecting..."
        state_probability = {}

        xml_root = dom.parse( xml_file )
        components = xml_root.getElementsByTagName( "Component" )
        for c in components:
            states = c.getElementsByTagName( "State" )
            for s in states:
                state_name = s.getAttribute( "Name" )
                p = get_child( s, "Probability" )
                if p != None:
                    prob_string = "%.1f" % ( float( p.firstChild.nodeValue ) * 100 )
                    state_probability[ state_name ] = ( prob_string + '%' )

        states = self.xmi_root.getElementsByTagName( "Behavioral_Elements.State_Machines.State" )
        for s in states:
            s_name = get_name( s )
            if s_name != "":
                prob = state_probability[ s_name ]
                if prob != "":
                    name_node = get_child( s, ".name" )
                    new_text_node = self.xmi_doc.createTextNode( s_name + " [" + prob + "]" )
                    name_node.replaceChild( new_text_node, name_node.firstChild )



    def clean( self ):
        print "Cleaning..."
        states = self.xmi_doc.documentElement.getElementsByTagName( "Behavioral_Elements.State_Machines.State" )
        for s in states:
            s_name = get_name( s )
            if s_name != "":
                name_node = get_child( s, ".name" )
                new_text_node = self.xmi_doc.createTextNode( s_name )
                name_node.replaceChild( new_text_node, name_node.firstChild )



    def effect_changes( self, new_file="" ):

        if new_file.endswith( ".zargo" ) and not self.xmi_file.endswith( ".zargo" ):
            print "Cannot write to "+new_file+". Writing to "+self.xmi_file+" instead."
```

```
    new_file = ""
if new_file == "":  new_file = self.xmi_file

print "Making changes to "+new_file

if self.xmi_file.endswith( ".zargo" ) and new_file.endswith( ".zargo" ):
    z = zipfile.ZipFile( self.xmi_file )
    z_new = zipfile.ZipFile( new_file, "w" )
    for f in z.namelist( ):
        info = z.getinfo( f )
        new_info = zipfile.ZipInfo()
        new_info.filename = info.filename
        new_info.date_time = info.date_time
        new_info.compress_type = zipfile.ZIP_DEFLATED
        if f.endswith( ".xmi" ):  # write new xmi file
            z_new.writestr( new_info, self.xmi_root.toxml( ) )
        else:  # copy old file with (minimal) ZipInfo header
            z_new.writestr( new_info, z.read( f ) )

    z.close( )
    z_new.close( )


else:
    modified_xmi_file = open( new_file, "w" )
    print >> modified_xmi_file, self.xmi_root.toxml( )
    modified_xmi_file.close( )
```

C.7 extract script

```
#!/usr/bin/env /home/ccanevet/local/bin/python2.1
# extract

import sys
from PEPA_Extractor import PEPA_Extractor


def usage( arg ):
        print "error: "+arg
        print "Usage: extract [-n] [-q] [-p=<output_file>] <input_file>"
        print "<file> must be of type .xmi or .zargo"
        sys.exit(1)


def get_file_to_parse( ):
    file = raw_input( "Please enter the name of the file: " )
    if file.startswith("\"") and file.endswith("\""):
        file = file[ 1 : -1 ]
    return file



enable_cache = 1
input_file = ""
output_file = ""
print_output = 1
write_output = 1


max = len( sys.argv )
for index in range( max ):
        arg = sys.argv[ index ]

        if arg.endswith( "extract" ):
                continue

        elif arg.startswith( "-p" ) or arg.startswith( "--pepa-file" ):
                output_file = arg[ 3 : len( arg ) ]

        elif ( arg == "--no-pepa") or ( arg == "-n" ):
                write_output = 0

        elif ( arg == "--quiet") or ( arg == "-q" ):
                print_output = 0

        elif arg.endswith(".xmi") or arg.endswith(".zargo"):
                input_file = arg

        else:
                usage( arg )


e = PEPA_Extractor( )
if input_file == "":
        input_file = get_file_to_parse( )

print
e.parse( input_file )
e.generate_PEPA( )
if print_output:
        e.print_PEPA_to_screen( )
if write_output:
        e.write_PEPA_to_file( output_file )
```

C.8 reflect script

```
#!/usr/bin/env /home/ccanevet/.majp/Python-2.2.1/python
# reflect

import sys
from Reflector import Reflector


xml_file = ""
xmi_file = ""
output_file = ""
clean_xmi = 0

def usage( arg ):
        print "error: "+arg
        print "Usage: reflect [-c|--clean] [-n|--new=new_file] [xml_file] [xmi_file]"
        sys.exit(1)


def get_file_to_parse( string="" ):
    file = raw_input( "Please enter the name of " + string + " file: " )
    if file.startswith("\"") and file.endswith("\""):
        file = file[ 1 : -1 ]
    return file


for arg in sys.argv:

        if arg.endswith( "reflect" ):
                continue

        elif arg.startswith( "-n" ) or arg.startswith( "--new" ):
                output_file = arg[ arg.find( "=" ) + 1 : len( arg ) ]

        elif ( arg == "--clean" ) or ( arg == "-c" ):
                clean_xmi = 1

        elif arg.endswith(".xmi") or arg.endswith(".zargo"):
                xmi_file = arg

        elif arg.endswith(".xml"):
                xml_file = arg

        else:
                usage( arg )


if xmi_file == "":
        xmi_file = get_file_to_parse( ".xmi or .zargo" )

r = Reflector( )
r.parse( xmi_file )

if clean_xmi:
        r.clean( )
else:
        if xml_file == "":
                xml_file = get_file_to_parse( ".xml" )
        r.reflect( xml_file )


if output_file == "":
        output_file = xmi_file
r.effect_changes( output_file )
```