Documentation for Java Extractor / Reflector

Catherine Canevet, Stephen Gilmore and Matthew Prowse

November 25, 2002

Contents

1 Introduction

The ability for designers to capture performance related details of their systems at an early stage in development may help to highlight design errors before they become a problem. The software toolset described here permits the formal analysis of UML models to solve them for performance results, without requiring the modeller to fully understand the techniques used.

There are a number of UML modelling tools such as ArgoUML [arg], Poseidon [pos] or Rational Rose [ros]. In principle, any tool capable of exporting a UML model in XMI format can be used. The performance algebra PEPA [Hil96] has been chosen as an intermediate language in the performance analysis process; the solving of these models performed by the PEPA Workbench [GH94].

This work is part of the DEGAS [deg] project.

1.1 Structure of This Document

Section 2 describes the primary role of both the Extractor and Reflector, followed by discussion in Section 3 of the algorithms used to generate the PEPA model. Section 4 describes the current Java implementation. Section 5 gives details of how this implementation has been integrated with the existing PEPA Workbench.

In appendices A and B, there are two example extractions intended to demonstrate the algorithms given in Section 3. Appendix C contains the entire Java source code for both packages.

2 What Does It Do?

The software toolset allows UML modellers to annotate their models with performance information. An equivalent performance model is extracted from the UML, solved, and the results reflected back to the UML level.

2.1 Extractor

The Extractor takes as input an `.xmi` model or a `.zargo` file containing an `.xmi` model and from which, generates the equivalent performance model in PEPA. This PEPA model can then be written to a `.pepa` file and used as input to the PEPA Workbench.

2.2 Reflector

The Reflector takes as input the same `.xmi` or `.zargo` file used as input to the Extractor, and the `.xml` results file generated by the PEPA Workbench on solving the model. The original model is annotated with these results, and a modifed `.xmi` or `.zargo` file is generated.

3 How Does It Work?

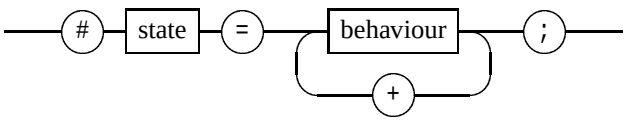
3.1 Extractor

The UML model used as input to the Extractor should consist of one or more classes each with a State Diagrams describing its behaviour, and a Collaboration Diagram showing associations between instances of these classes.

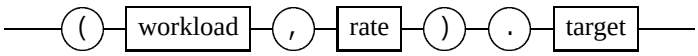
To generate a PEPA model, the Extractor extracts certain information from the UML model. The State Diagrams are used to generate *component definitions* and the Collaboration Diagram to generate the *system equation*.

3.1.1 Component Definitions

definition



behaviour



Each state in a State Diagram produces a *definition* consisting of a state and a number of *behaviours*. Each outgoing transition from the state corresponds to a *behaviour* consisting of a workload, a rate and a target state. (The workload and rate correspond to the event name and action expression of the transition respectively).

From the State Diagrams of the UML model, it is possible to generate these *component definitions* using a simple algorithm:

generate_definitions( )

```
 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 = behaviors0 [ + behaviors1 [ + ... ] ]"
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.

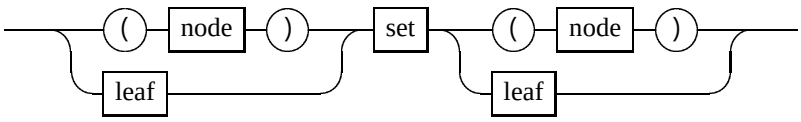
Lines 6-9 generate the behaviour corresponding to the current transition. The event name and action expression of the transition, and the name of the target state define this behaviour.

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

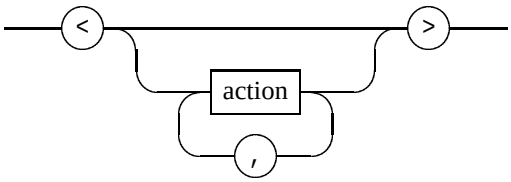
Line 15 returns the generated terms.

3.1.2 System Equation

node



set



leaf

A *node* represents the cooperation (or synchronisation) between two components. The *set* is a list of actions on which the two components should cooperate. A *leaf* is a component, given by the name of the state in which it should be initialised.

The work required to produce the *system equation* is two-fold. Firstly, a set of *atomic nodes* must be generated. (An *atomic node* is a *node* of depth one). Each association in the Collaboration Diagram produces an *atomic node*. Secondly, these *atomic nodes* must be merged to form just one. This is then the full *system equation*.

generate_system_equation( )

1: *cooperations* ← *empty list*
2: **for** each association *a* (of the collaboration diagram) **do**
3: $left\_instance \leftarrow$ left element of *a*
4: $right\_instance \leftarrow$ right element of *a*
5: $left\_sm \leftarrow$ statemachine of the class of $left\_instance$
6: $right\_sm \leftarrow$ statemachine of the class of $right\_instance$
7: $left\_leaf \leftarrow Leaf$(initial state of $left\_sm$, $left\_instance$)
8: $right\_leaf \leftarrow Leaf$(initial state of $right\_sm$, $right\_instance$)
9: $sync \leftarrow$ (events in $left\_sm$) ∩ (events in $right\_sm$)
10: $cooperations \leftarrow cooperations + Node(\ left\_leaf, sync, right\_leaf\ )$

```
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* loop 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 that 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*.

Line 17 uses a function called *insert*. The way this function performs is crucial to producing the correct system equation. The following algorithm appears to perform correctly for all small models tested so far, including those containing repeated components:

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, 1 )
```

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 *insert* 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.

It may be necessary to introduce *Hiding* into the *system equation* to ensure correct cooperation between components. There is no example yet of where this might be necessary.

4 How Is It Implemented?

4.1 The *extractor* Package

There are two primary classes in the *extractor* package: *Extractor* and *PEPA_Extractor*.

4.1.1 Extractor

The class *Extractor* performs file handling, DOM parsing, and provides convenient methods to make certain information in the DOM tree more accessible.

The *parse()* method takes an object of class *File*. (If a `.zargo` file, it will locate the `.xmi` file within the archive using the *java.util.zip* package). The `.xmi` file is parsed using the *javax.xml.parsers.DocumentBuilder* DOM parser resulting in a *Document* object put into *dom_doc*. The file will be put into *xmi_file*. A successful parse will return `true`, otherwise `false` will be returned.

The *getElement()* method will return the *Element* from the DOM tree whose `xmi.id` attribute matches the given string value. This can be quite time consuming, and so a cache table is maintained called *dom_element_cache*.

The *getName()* static method locates the "name" child of the specified node. The value of this node is returned. (Any existing performance annotations are ignored).

The *getChild()* static method will return the child of the specified node, whose tag name matches the given string value.

The *getByRef()* method uses *getChild()* to locate the named child of the specified node. The child of this node will have an attribute `xmi.idref`. Using the *getElement()* method, the node with the matching `xmi.id` attribute is returnd. The *getAllByRef()* method returns all such referenced nodes.

The *getOwned()* static method uses *getChild()* to locate the "ownedElement" child of the specified node. All children of this child are returned.

4.1.2 PEPA_Extractor

The class *PEPA_Extractor* is a subclass of the class *Extractor*. It extracts from the DOM tree the information needed to generate a corresponding PEPA model. As it does so, an abstract syntax is generated which is then converted into a concrete, string representation.

There are a number of methods used to locate specific elements in the model or parts of it, in particular State Diagrams and Collaboration Diagrams. Although important, these do not affect the generation of the PEPA model. (One point worth noting is that the return type of *getAssociations()*, a 2-dimensional array of *Element* objects, represents an array of pairs of associated *Elements*). The methods described below are used in the generation of the PEPA model.

The *generatePEPA()* method makes three calls to *getPEPA_definitions()*, *getPEPA_rates()* and *getPEPA_cooperation()* in that order and returns an array of strings. Although the rates appear before the definitions in the output, they are generated by *getPEPA_definitions()* and so the order in which they are called matters.

The *getPEPA_definitions()* method converts to string representation the *Definition* objects returned by *generatePEPA_definitions()* which takes a single `StateMachine` *Element* object. For each `State` *Element* in the state machine, *generatePEPA_definitions()* produces a *Definition* object, composed of objects from the *pepa.process* package. The behaviours in a definition are sorted on target state, and the list of definitions corresponding to a state machine are

sorted with the initial state first.

The *getPEPA_rates()* method produces a number of rate variable assignments, initialising to a default value all rate variables encountered while producing the component behaviours.

The *getPEPA_cooperation()* method produces a single PEPA cooperation component, composed of the atomic cooperations created by *generatePEPA_cooperations()* from the associations in the Collaboration Diagram. There are two classes used to represent cooperations: *CoopNode* and *CoopLeaf*. An object of the *CoopLeaf* class has two field values determining its identity: the components initial `State` *Element* object and an *Element* representing the `ClassifierRole` (a particular instance of a class). There is a difference worth noting between the two *contains()* methods in both classes. One takes a *CoopLeaf* object as an argument, the other an `State` *Element* object. The former will return `true` iff the *CoopNode* or *CoopLeaf* on which it is called contains an exact match, or is itself an exact match of the given instance. The latter will return `true` if there is another instance of the same class present. This makes it possible to successfully insert repeated components.

The *writePEPA()* method calls the *generatePEPA()* method, and writes the model that is returned to a `.pepa` file. The filename is determined from the input filename, simply replacing the `.xmi` or `.zargo` with `.pepa` and writing the file to the same directory.

4.2 The *reflector* Package

There is one class in this package called *Reflector*. This simple class has two static fields which store the original `.xmi` or `.zargo` file that was parsed using the *PEPA_Extractor*, and the `.xml` file that holds the results generated by the PEPA Workbench.

The static method *reflect()* then performs the reflection. The two files are parsed using the *parse()* method of the *Extractor* class, the parsed *Document* objects being returned using its *getDocument()* method. The results contained within the `.xml` file are extracted to a hashtable containing probabilities indexed by the states they represent. The `.xmi` model is then updated with the results by annotating each state name with the probability associated to it. (The assumption is made that all state names are unique. PEPA would not produce correct results otherwise).

The modified model is then written back to file. If the original file had the suffix `.xmi`, the suffix of the reflected model will be `.reflected.xmi`. Similarly with a `.zargo` file. This ensures the original model is not actually altered.

5 How It All Fits Together

These two packages become *pepa.extractor* and *pepa.reflector* respectively. The two methods of invocation either instantiate the *pepa.gui.jpwb* class or the *pepa.tty.JPWBtty* class. The former is an interactive version of the workbench, which required that models be loaded manually. The latter performs the loading and solving at the command line.

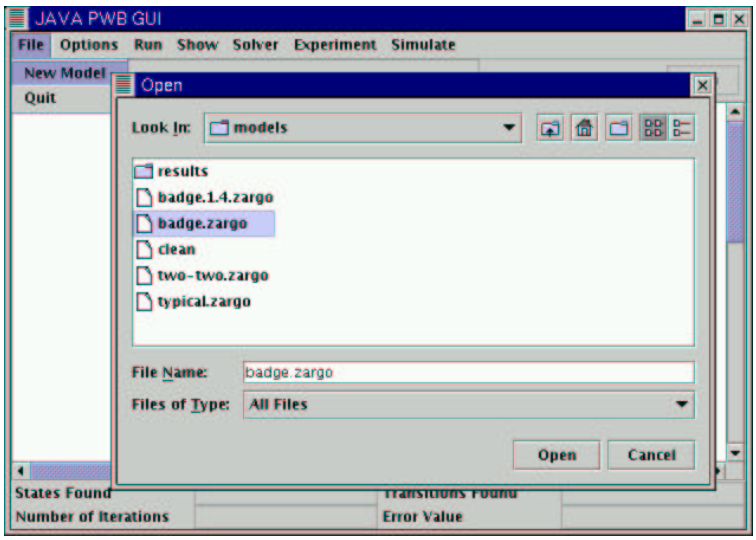
In both cases, before the point at which the model is loaded in the method *loadmodel()*, before the filename is passed to *Peparoni*, a check is performed to determine if the input file is an `.xmi` or a `.zargo` file. If it is indeed one of these, the *PEPA_Extractor* is instantiated, the model parsed, and the PEPA model generated. The filename of this new `.pepa` file is then passed on for loading into the PEPA workbench.

When the *PEPA_Extractor* successfully parses a model, the *File* object is passed to the *Reflector* using its static *setOriginal()* method. Similarly, when the PEPA workbench solves a model and produces an `.xml` results file, a *File* object is passed to the *Reflector* using its static *setResults()* method. The latter call is performed by *solve()* method of the *pepa.pepa.Peparoni* class.

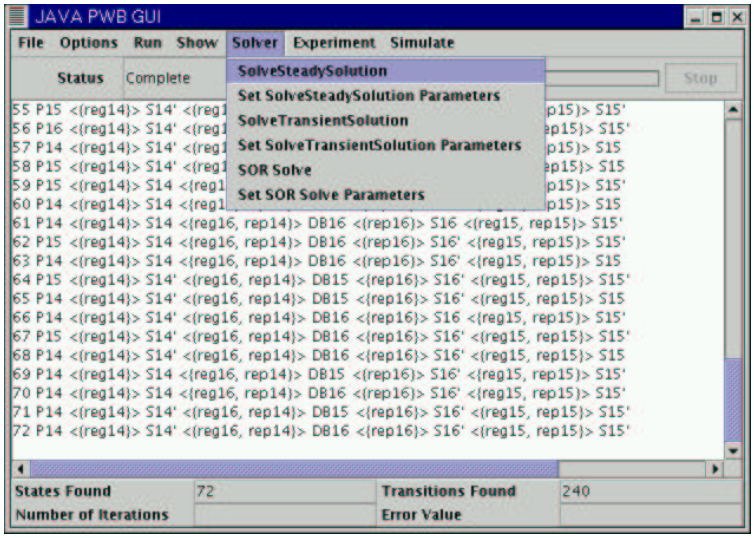
Once the *Reflector* has the two files it needs to proceed with reflection, the static *reflect()* method is called, again by the *solve()* method of the *pepa.pepa.Peparoni* class. If reflection is not intended to occur, at least one of the *File* objects will remain at its default value of *null* and the method will fail safe.

6 What It Looks Like In Action

Both *Extractor* and *Reflector* integrate seamlessly with the existing PEPA Workbench.



It is possible to load either a `.xmi` or a `.zargo` file directly into the PEPA Workbench. The `.pepa` file is generated and loaded in one step, without the need for intervention.



After the model has been run, solving the model to steady state solution will generate the modified `.xmi` or a `.zargo` file in the same directory from which it was loaded.

References

[arg] Argouml, a java based cognitive case tool. . http://argouml.tigris.org/.

[CGH99] G. Clark, S. Gilmore, and J. Hillston. Specifying performance measures for PEPA. In J.-P. Katoen, editor, *Proceedings of the Fifth International AMAST Workshop on Real-Time and Probabilistic Systems*, number 1601 in LNCS, pages 211–227, Bamberg, Germany, May 1999. Springer-Verlag.

[deg] Design environments for global applications . http://www.omnys.it/degas/.

[GH94] S. Gilmore and J. Hillston. The PEPA Workbench: A Tool to Support a Process Algebra-based Approach to Performance Modelling. In *Proceedings of the Seventh International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, number 794 in Lecture Notes in Computer Science, pages 353–368, Vienna, May 1994. Springer-Verlag.

[Hil96] J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.

[pos] Poseidon for uml, a fully-fledged uml case tool. . http://www.gentleware.com/products/.

[ros] Rational rose, a model-driven development tool. . http://www.rational.com/products/rose/.

A Example: Active Badge

Here is shown a worked example taken from [CGH99]. It is intended to demonstrate the algorithms presented in Section 3. (Producing the definitions is a trivial task. This process has not been shown in any detail).

Generating the Terms

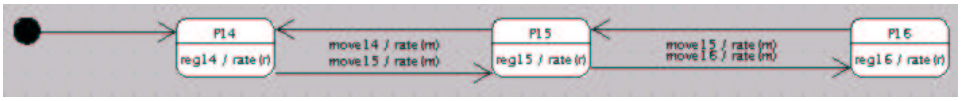


Figure 1: The Person state diagram

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;

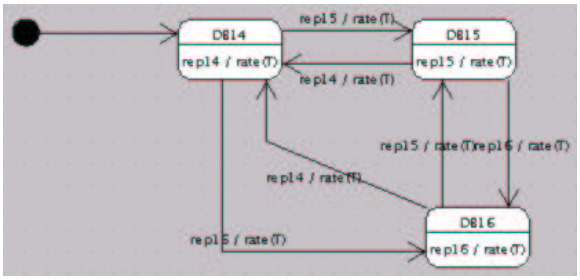


Figure 2: The Database state diagram

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;

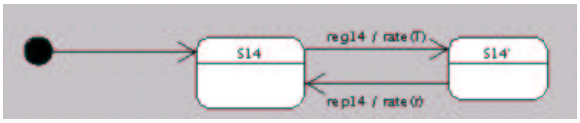


Figure 3: The Sensor S14 state diagram

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

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

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

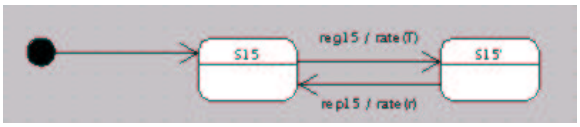


Figure 4: The Sensor S15 state diagram

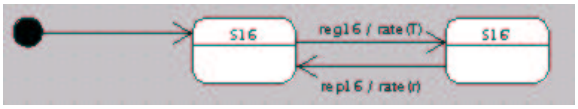


Figure 5: The Sensor S16 state diagram

Generating the System Equation

Here is the collaboration diagram for this example :

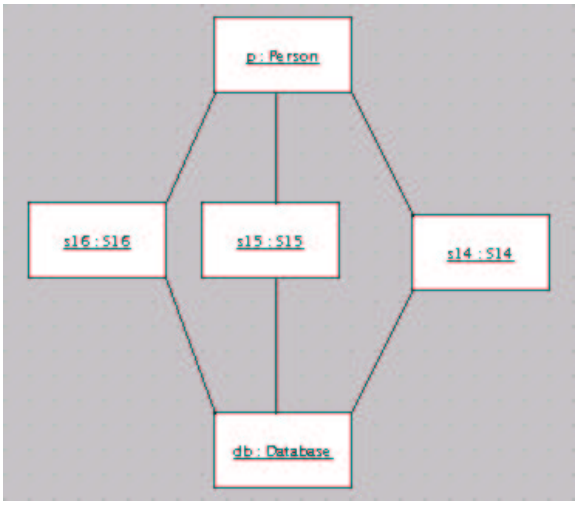


Figure 6: The collaboration diagram

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:

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

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)

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

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):

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

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

Here is a second worked example. It demonstrates fully the algorithm used to generate the system equation. (Again the process of generating the definitions is trivial and has not been shown in detail).

Generating the Terms

Figure 7: The User state diagram

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

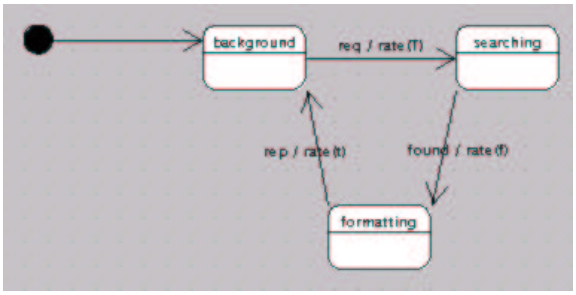


Figure 8: The Server state diagram

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

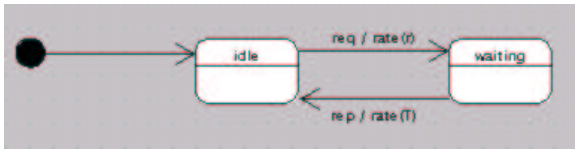


Figure 9: The Client state diagram

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

Generating the System Equation

Here is the collaboration diagram for this example :

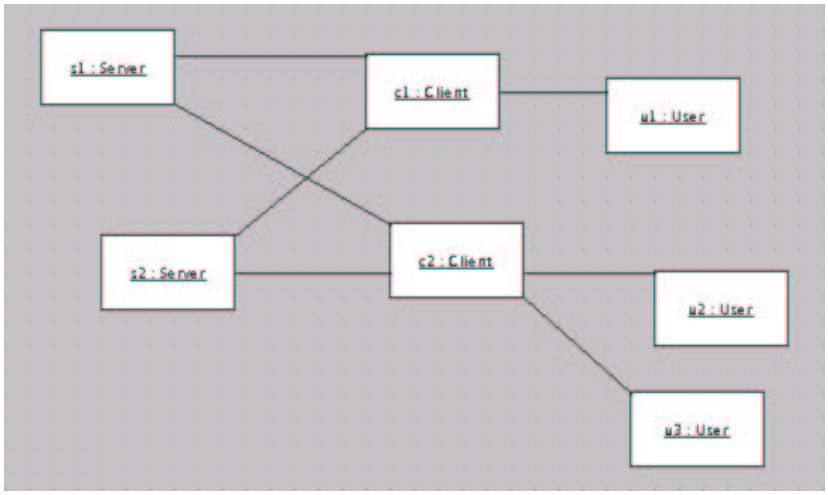


Figure 10: The collaboration diagram

The atomic cooperations are the following :

$$S1 \underset{req,rep}{\bowtie} C1 \quad (7)$$

$$S2 \underset{req,rep}{\bowtie} C2 \quad (8)$$

$$S1 \underset{req,rep}{\bowtie} C2 \quad (9)$$

$$U3 \underset{req}{\bowtie} C2 \quad (10)$$

$$U1 \underset{req}{\bowtie} C1 \quad (11)$$

$$S2 \underset{req,rep}{\bowtie} C1 \quad (12)$$

$$U2 \underset{req}{\bowtie} C2 \quad (13)$$

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 $\bowtie$ (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)** $\bowtie$ **(U3** $\underset{req}{\bowtie}$ **C2))**

Step 6 : Try to insert (6)

top : S1 $\underset{req,rep}{\bowtie}$ ((C1 $\underset{req}{\bowtie}$ U1) $\bowtie$ (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 $\bowtie$ S2

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

Step 7 : Try to insert (7)

top : (S1 $\bowtie$ S2) $\underset{req,rep}{\bowtie}$ ((C1 $\underset{req}{\bowtie}$ U1) $\bowtie$ (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) $\bowtie$ (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

(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 :

(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 ) )
```

A Program Listings

A.1 Extractor.java

```
package pepa.extractor;

import java.io.*;
import java.util.*;
import java.util.zip.*;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;

import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;

/**
 * The <code>Extractor</code> class is a base class for extracting information from <code>XMI</code> models.
 * <p>
 * It provides functions such as file handling, <code>Element</code> lookup by <code>xmi.id</code> attribute value
 * and commonly used patterns to extract <code>Element</code> objects from the model.
 * <p>
 * It is implemented by parsing the <code>XMI</code> model using the <code>javax.xml.parsers.DocumentBuilder</code>
 * DOM parser.
 */
public class Extractor {

    /** A DOM parser */
    private DocumentBuilder builder;

    /** <code>Elements</code> indexed by their <code>xmi.id</code> attribute value */
    private Hashtable dom_element_cache;

    /** The parsed <code>Document</code> */
    Document dom_doc;

    /** The <code>Element</code> that represents the <code>Document</code> content */

    /** The <code>File</code> that has been parsed */
    File xmi_file;

    /**
     * Initialises a new <code>Extractor</code> object.
     * <p>
     * The DOM parser is instantiated.
     * <p>
     * @see Extractor#Extractor( boolean b )
     */
    public Extractor() {
        try {
            DocumentBuilderFactory factory =
                DocumentBuilderFactory.newInstance();
            this.builder = factory.newDocumentBuilder();
        } catch (Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
        this.dom_element_cache = new Hashtable();
        this.xmi_file = null;
    }


    /**
     * <code>getter</code> method for returning the <code>Document dom_doc</code> field.
     * <p>
     * Called by the Reflector when using the Extractor for parsing
     * <p>
     * @return the <code>Document</code> object
     */
    public Document getDocument() { return this.dom_doc; }


    /**
     * Locates all <code>Element</code> objects pointed to by the children of a named child <code>Element</code>
     * of the specified <code>Element</code> object.
     * <p>
     * @param root an <code>Element</code> object
     * @param name the name of the child
     * @return the referenced <code>Element</code> objects - may return []
     */
    public Element[] getAllByRef(Element root, String name) {
        ArrayList result = new ArrayList();
        if (root != null) {
            Element named_child = getChild(root, name);
            if (named_child != null) {
                NodeList children = named_child.getChildNodes();
                for (int i = 0; i < children.getLength(); i++) {
                    Node child = children.item(i);
                    if (child instanceof Element) {
                        String idref =
                            ((Element) child).getAttribute("xmi.idref");
                        result.add(this.getElement(idref));
                    }
                }
            }
        }
        return (Element[]) result.toArray(new Element[result.size()]);
    }


    /**
```

```
     * Locates the <code>Element</code> objects pointed to by the (first) child of a named child
     * <code>Element</code> of the specified <code>Element</code> object.
     * <p>
     * @param root an <code>Element</code> object
     * @param name the name of the child
     * @return the (first) referenced <code>Element</code> object - may return null
     */
    public Element getByRef(Element root, String name) {
        Element[] all = this.getAllByRef(root, name);
        if (all.length > 0)
            return all[0];

        // Print diagnostic error message before returning null
        Diagnostics.say("Extractor.getByRef called with \"" + name + "\", returning null");
        return null;
    }

    /**
     * Locates the named child <code>Element</code> of the specified <code>Element</code> object.
     * <p>
     * @param root an <code>Element</code> object
     * @param name the name of the required child
     * @return the child <code>Element</code> object - may return null
     */
    public static Element getChild(Element root, String name) {
        if (root != null) {
            NodeList children = root.getChildNodes();
            for (int i = 0; i < children.getLength(); i++) {
                Node child = children.item(i);
                if (child instanceof Element)
                    if (((Element) child).getTagName().endsWith(name))
                        return (Element) child;
            }
        }

        // Print diagnostic error message before returning null
        if (root == null) {
            Diagnostics.say("Extractor.getChild called with null pointer, returning null");
        } else {
            Diagnostics.say("Extractor.getChild failed on " + name);
        }
        return null;
    }


    /**
     * Locates an <code>Element</code> object whose <code>xmi.id</code> attribute matches the specified value.
     * <p>
     * @param id the required value of the <code>xmi.id</code> attribute
     * @return the corresponding <code>Element</code> object - may return null
     */
    public Element getElement(String id) {
        if (!this.dom_element_cache.containsKey(id)) {
            Element elem = this.getElement_helper(this.dom_doc.getDocumentElement(), id);
            if (elem != null)
                this.dom_element_cache.put(id, elem);
            else
                return null;
        }
        return (Element) this.dom_element_cache.get(id);
    }


    /**
     * A helper method for <code>getElement</code>.
     * <p>
     * Recursively performs the search on the children of the specified <code>Element</code> object.
     * <p>
     * @param root a <code>Element</code> object
     * @param id the required value of the <code>xmi.id</code> attribute
     * @return the corresponding <code>Element</code> object - may return null
     */
    private Element getElement_helper(Element root, String id) {
        NodeList children = root.getChildNodes();
        for (int i = 0; i < children.getLength(); i++) {
            Node child = children.item(i);
            if (child instanceof Element) {
                Element c = (Element) child;
                if (c.hasAttribute("xmi.id")) {
                    String this_id = c.getAttribute("xmi.id");
                    if (this_id.equals(id))
                        return c;
                    this.dom_element_cache.put(this_id, c);
                }
                Element result = this.getElement_helper(c, id);
                if (result != null)
                    return result;
            }
        }
        return null;
    }


    /**
     * Locates the <code>name Element</code> of the specified <code>Element</code> and returns its value.
     * <p>
     * @param node an <code>Element</code> object
     * @return the value of its <code>name Element</code> - may return ""
     */
    public static String getName(Element node) {
        Element name_node = getChild(node, ".name");
        if (name_node == null)
            return "";
        if (name_node.getFirstChild() == null)
            return "***null***";
        String name = name_node.getFirstChild().getNodeValue();
```

```
        if (name.indexOf("[") != -1)
            name = name.substring(0, name.indexOf("[") - 1);
        return name;
    }


    /**
     * Locates all children <code>Element</code> objects of the <code>ownedElement</code> child of the specified <code>Element</code> object.
     * <p>
     * @param node an <code>Element</code> object
     * @return all <code>ownedElement</code> children <code>Element</code> objects
     */
    public static Element[] getOwned(Element node) {
        ArrayList result = new ArrayList();
        if (node != null) {
            Element ownedElement = getChild(node, ".ownedElement");
            if (ownedElement != null) {
                NodeList children = ownedElement.getChildNodes();
                for (int i = 0; i < children.getLength(); i++) {
                    Node child = children.item(i);
                    if (child instanceof Element)
                        result.add(child);
                }
            }
        }
        return (Element[]) result.toArray(new Element[result.size()]);
    }


    /**
     * Parses the specified <code>File</code> with a DOM parser.
     * <p>
     * If the specified <code>File</code> object is an <code>.xmi</code> file, it can be parsed directly.
     * If the specified <code>File</code> object is a <code>.zargo</code> file, the contained <code>.xmi</code>
     * file must be located and extracted.
     * <p>
     * If the <code>as_utility</code> flag in this class is not set, then the <code>Reflector</code> class
     * is passed the input <code>File</code> object for later reflection of results from the PEPA workbench.
     * <p>
     * @param f a <code>File</code> object
     * @return <code>true</code> for a successful parse, <code>false</code> otherwise
     */
    public boolean parse(File f) {
        if( f.exists() ) {
            if( f.getName().toLowerCase().endsWith( ".xmi" ) || f.getName().toLowerCase().endsWith( ".xml" ) ) {
                try {
                    this.xmi_file = f;
                    this.dom_doc = this.builder.parse(f);
                    return true;
                } catch (Exception e) {
                    e.printStackTrace();
                    return false;
                }
            } else if (f.getName().toLowerCase().endsWith(".zargo")) {
                try {
                    ZipInputStream zip_is =
                        new ZipInputStream(new FileInputStream(f));
                    ZipEntry entry = zip_is.getNextEntry();
                    while (entry != null) {
                        if (entry.getName().toLowerCase().endsWith(".xmi")) {
                            this.xmi_file = f;
                            this.dom_doc = this.builder.parse(zip_is);
                            break;
                        }
                        zip_is.closeEntry();
                        entry = zip_is.getNextEntry();
                    }
                    zip_is.close();
                } catch (Exception e) {
                    e.printStackTrace();
                    return false;
                }
                if (this.dom_doc != null)
                    return true;
            }
        }
        return false;
    }
}
```

A.2 PEPA_Extractor.java

```
package pepa.extractor;

import java.io.*;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Hashtable;
import java.util.HashSet;

import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;

import pepa.process.Process;
import pepa.xml.XMLresults;

/**
 * The <code>PEPA_Extractor</code> class extends the <code>Extractor</code> class, converting UML models to PEPA models.
 * <p>
 * It extracts information from the State and Collaboration diagrams,
 * creating abstract syntax objects whose <code>toString()</code> methods produce
 * PEPA syntax which can then be loaded by the PEPA workbench.
 */
public class PEPA_Extractor extends Extractor {

    /** A two dimensional array consisting of 'pairs' of <code>ClassifierRole Element</code> objects. */
    private Element[][] associations;

    /** Arrays of <code>State Element</code> objectss indexed by the <code>StateMachine Element</code> in which they are found. */
    private Hashtable states;

    /** Arrays of <code>String</code> objects indexed by the <code>StateMachine Element</code> in which they are present */
    private Hashtable actions;

    /** <code>State Element</code> objects indexed by the <code>StateMachine Element</code> in which they are the initial states. */
    private Hashtable initial;

    /** <code>StateMachine Element</code> objects indexed by the <code>Classifier Element</code> objects they are owned by. */
    private Hashtable statemachines;

    /** An <code>ArrayList</code> of <code>String</code> elements representing the rate variables used in the model */
    private ArrayList rates;


    /** Initialises a new <code>PEPA_Extractor</code> object.
     * <p>
     * Indicates to the <code>XMLresults</code> class that XML results should be generated.
     *
     * @see pepa.extractor.Extractor#Extractor()
     * @see pepa.xml.XMLresults#resultsWanted()
     */
    public PEPA_Extractor() {
        super();
        XMLresults.resultsWanted();
    }

    /**
     * Main method.
     * <p>
     */
    public static void main(String[] args) {
        PEPA_Extractor e = new PEPA_Extractor();
        String filename = "";


        if (args == null)
            Diagnostics.fatalError("No file name supplied");
        if (args.length == 2) {
            if (!args[0].equals("-verbose"))
                Diagnostics.fatalError("Usage: java pepa.extractor.PEPA_Extractor" +
                                       " [-verbose] filename.{xmi|zargo}");
            Diagnostics.verbose = true;
            Diagnostics.say("PEPA extractor [Version of 14/11/2002]");
            Diagnostics.say("Verbose output selected");
            filename = args[1];
        } else {
            if (args.length != 1)
                Diagnostics.fatalError("Usage: java pepa.extractor.PEPA_Extractor" +
                                       " [-verbose] filename.{xmi|zargo}");
            filename = args[0];
        }

        Diagnostics.say("Starting extractor");
        Diagnostics.say("Loading file");
        File f = new File(filename);
        if (!e.parse(f))
            Diagnostics.fatalError("File failed to parse");

        Diagnostics.say("File parsed successfully");
        pepa.process.Action.initialise();
        Diagnostics.say("Starting to form PEPA output");
        String pepaFileName = e.writePEPA();
        Diagnostics.say("Output formed successfully");
        Diagnostics.say("Output written to file :");
        Diagnostics.say("\t" + pepaFileName);
        if (Diagnostics.warnings == 0) {
            Diagnostics.say("Exiting extractor successfully");
        }
        else {
            Diagnostics.verbose = true;
            Diagnostics.say("\nThe PEPA Extractor reported " + Diagnostics.warnings + " warnings");
        }
    }
```

```
/**
 * Initialises all instance fields (cache tables) and parses.
 * @see pepa.extractor.Extractor#parse(File)
 */
public boolean parse(File f) {
    if (f == null)
        return false;

    this.associations = null;
    this.states = new Hashtable();
    this.actions = new Hashtable();
    this.initial = new Hashtable();
    this.statemachines = new Hashtable();
    this.rates = new ArrayList();

    boolean succ = super.parse(f);

    // Tell the Reflector which file to reflect to.
    if( succ ) pepa.reflector.Reflector.setOriginal( f );

    return succ;
}

/**
 * Generates and converts to string representation the PEPA component definitions.
 * <p>
 * @return All definitions.
 *
 * @see PEPA_Extractor#generatePEPA_definitions
 * @see PEPA_Extractor#generatePEPA_definition
 */
private String[] getPEPA_definitions() {
    ArrayList result = new ArrayList();

    Element[] state_machines = this.getStateMachines();
    for (int i = 0; i < state_machines.length; i++) {

        Definition[] def_list =
            this.generatePEPA_definitions(state_machines[i]);
        result.addAll(Arrays.asList(def_list));
        result.add("");
    }
    String[] strings = new String[result.size()];
    for (int i = 0; i < result.size(); i++) {
        if (result.get(i) == null) {
            Diagnostics.warning("The PEPA Extractor found a null pointer in the results while.");
            Diagnostics.diagnostic("processing the PEPA component definitions.  This may have");
            Diagnostics.diagnostic("been caused by a previously reported error.  Will insert");
            Diagnostics.diagnostic("the string ***null*** and attempt to continue.");
            strings[i] = "***null***";
        } else {
            strings[i] = result.get(i).toString();
        }
    }
    return strings;
}

/**
 * Generates abstract syntax <code>Definition</code> objects from a <code>StateMachine Element</code> object.
 * <p>
 * These correspond to the PEPA 'component' represented by the states and transitions of
 * the specified statemachine.
 *
 * <p>
 * @param statemachine a <code>StateMachine Element</code> object.
 * @return The corresponding <code>Definition</code> objects.
 */
private Definition[] generatePEPA_definitions(Element statemachine) {
    ArrayList this_sm = new ArrayList();
    Element[] states = this.getStates(statemachine);
    for (int j = 0; j < states.length; j++) {
        Definition def = generatePEPA_definition(states[j]);
        if (def != null)
            this_sm.add(def);
    }
    Definition[] def_list =
        (Definition[]) this_sm.toArray(new Definition[this_sm.size()]);
    def_list =
        Definition.sort(
                        new Process.Var(getName(this.getInitialState(statemachine))),
                        def_list);
    //Arrays.sort( def_list, new DefComparator( initial, def_list ) );
    return def_list;
}

/**
 * Generates an abstract syntax <code>Definition</code> object from a specified <code>State Element</code> object.
 * <p>
 * This <code>Definition</code> represents the behaviours the component can exhibit when in the current state.
 *
 * <p>
 * @param state a <code>State Element</code> object.
 * @return The representative <code>Definition</code> object
 */
private Definition generatePEPA_definition(Element state) {
    if (state.getTagName().endsWith(".Pseudostate"))
        return null;

    // pepa.process.Process.Var
    Process v = new Process.Var(Extractor.getName(state));

    ArrayList results = new ArrayList();
    Element[] outgoing = this.getAllByRef(state, ".outgoing");
    for (int i = 0; i < outgoing.length; i++) {

        String action =
```

```
            Extractor.getName(this.getByRef(outgoing[i], ".trigger"));

        String rate = this.getRate(outgoing[i]);
        if (rate.equals("T"))
            rate = "infty";
        else if (!this.rates.contains(rate))
            this.rates.add(rate);
        pepa.rate.Rate r = pepa.rate.Rate.Strung(rate);

        pepa.process.Activity a = new pepa.process.Activity(action, r);

        String target =
            Extractor.getName(this.getByRef(outgoing[i], ".target"));
        Process p = new Process.Prefix(a, new Process.Var(target));

        results.add(p);
    }

    Process newP = null;

    Process[] list =
        (Process[]) results.toArray(new Process[results.size()]);
    Arrays.sort(list, new PrefixComparator());

    for (int i = 0; i < list.length; i++)
        if (newP == null)
            newP = list[i];
        else
            newP = new Process.Sum(newP, list[i]);

    return new Definition(v, newP);
}

/**
 * Generates and converts to string representation the component cooperations.
 * <p>
 * An array of <code>CoopNode</code> objects are merged to form one larger <code>CoopNode</code> object.
 * This is then converted to PEPA string representation.
 *
 * <p>
 * @return The string representation of the 'system equation'
 */
private String getPEPA_cooperation() {
    //              Cooperation[] coops = this.generatePEPA_cooperations();
    Diagnostics.say ("Extered PEPA_Extractor.getPEPA_cooperation");
    CoopNode[] coops = this.generatePEPA_cooperations();
    Diagnostics.say ("PEPA_Extractor.generatePEPA_cooperations returned an array of length " + coops.length);

    CoopNode C = null;
    //              Cooperation C = null;
    for (int i = 0; i < coops.length; i++) {
        //                  Cooperation c = (Cooperation) coops[i];
        CoopNode c = coops[i];
        if (C == null)
            C = c;
        else {
            Diagnostics.say ("Calling CoopNode.insert with \n\t" + C.toString());
            Diagnostics.say ("and \n\t" + c.toString());
            Diagnostics.say ("");
            C = CoopNode.insert(C, c);
        //                  else C = Cooperation.insert( C, c );
            }
    }
    if (C == null) {
        Diagnostics.warning("The PEPA Extractor found a null pointer in the results while.");
        Diagnostics.diagnostic("processing the cooperations in the PEPA system equation.");
        Diagnostics.diagnostic("This may have been caused by a previously reported error.");
        Diagnostics.diagnostic("Will insert the string ***null*** and attempt to continue.");

        return "***null***";
    } else {
        return C.toString();
    }
}

/**
 * Generates abstract syntax <code>CoopNode</code> objects.
 * <p>
 * Each <code>CoopNode</code> object corresponds to a single association in the Collaboration diagram
 * of the UML model. Each node is created from two <code>CoopLeaf</code> objects that represents
 * the states in which the cooperating components will be initialised, and a list of actions on which
 * they will cooperate (synchronise).
 *
 * <p>
 * @return All <code>CoopNode</code> objects.
 */
private CoopNode[] generatePEPA_cooperations() {
    Diagnostics.say ("Entered PEPA_Extractor.generatePEPA_cooperations");
    if (this.statemachines.size() == 0)
        this.getStateMachines();
    ArrayList coops = new ArrayList();
    Element[][] associations = this.getAssociations();
    Diagnostics.say ("PEPA_Extractor.getAssociations returned an array of length " + associations.length);

    for (int i = 0; i < associations.length; i++) {
        Element[] two = associations[i];

        Element a_sm =
            (Element) this.statemachines.get(
                                            this.getByRef(two[0], ".base"));
        Element b_sm =
            (Element) this.statemachines.get(
                                            this.getByRef(two[1], ".base"));

        Element a_state = this.getInitialState(a_sm);
        Element b_state = this.getInitialState(b_sm);
```

```
        //                  Instance a_inst = new Instance( Extractor.getName( a_state ), two[0] );
        //                  Instance b_inst = new Instance( Extractor.getName( b_state ), two[1] );

        CoopLeaf a_inst = new CoopLeaf(a_state, two[0]);
        CoopLeaf b_inst = new CoopLeaf(b_state, two[1]);

        String[] a_actions = this.getActions(a_sm);
        String[] b_actions = this.getActions(b_sm);

        HashSet intersection = new HashSet(Arrays.asList(a_actions));
        intersection.retainAll(Arrays.asList(b_actions));
        String[] actions =
            (String[]) intersection.toArray(
                                        new String[intersection.size()]);

        CoopNode c = new CoopNode(a_inst, actions, b_inst);
        coops.add(c);
        //                  coops.add( new Cooperation( a_inst, new CoopSet( new String[0] ), b_inst ) );
    }
    Object[] objects = coops.toArray(new CoopNode[coops.size()]);
    Arrays.sort((CoopNode[])objects);
    return (CoopNode[]) objects;
}


/**
 * Creates a list of all actions which a specified <code>StateMachine Element</code> admits.
 * <p>
 * The <code>actions</code> hashtable is filled the first time this method is called.
 * Subsequent calls need only return look up the correct list in this hashtable.
 * The list is a set - no elements are repeated.
 * <p>
 * @return The names of all actions.
 */
private String[] getActions(Element statemachine) {
    if (!this.actions.containsKey(statemachine)) {
        ArrayList result = new ArrayList();
        Element[] states = this.getStates(statemachine);
        for (int i = 0; i < states.length; i++) {
            Element[] outgoing = this.getAllByRef(states[i], ".outgoing");
            for (int j = 0; j < outgoing.length; j++) {
                String action =
                    this.getName(this.getByRef(outgoing[j], ".trigger"));
                if (!action.equals("") && !result.contains(action))
                    result.add(action);
            }
        }
        this.actions.put(
                        statemachine,
                        (String[]) result.toArray(new String[result.size()]));
    }
    return (String[]) this.actions.get(statemachine);
}

/**
 * Gets the rate from a specified <code>Transition Element</code> object.
 * <p>
 * A transition will have a rate associated with the action it performs.
 * It will generally be of the form "rate(x)" where 'x' is either a variable name or a real value.
 * Only the variable name or value is returned.
 *
 * <p>
 * @param trans a <code>Transition Element</code> object
 * @return The rate
 *
 */
private static String getRate(Element trans) {
    NodeList body =
        trans.getElementsByTagName("Foundation.Data_Types.Expression.body");
    String rate = body.item(0).getFirstChild().getNodeValue();
    if (rate.startsWith("rate("))
        rate = rate.substring(5, rate.length() - 1);
    return rate;
}

/**
 * Creates a list of all <code>StateMachine Element</code> objects present in the model.
 * <p>
 * The <code>statemachines</code> Hashtable is filled the first time this method is called.
 * Further calls need only return the values stored in the Hashtable.
 *
 * <p>
 * @return All <code>StateMachine Element</code> objects.
 */
private Element[] getStateMachines() {
    if (this.statemachines.size() == 0) {
        String STATEMACHINE =
            "Behavioral_Elements.State_Machines.StateMachine";
        NodeList state_machines =
            this.dom_doc.getDocumentElement().getElementsByTagName(STATEMACHINE);
        for (int i = 0; i < state_machines.getLength(); i++) {
            Element state_machine = (Element) state_machines.item(i);
            if (state_machine.hasAttribute("xmi.id")) {
                Element context = this.getByRef(state_machine, ".context");
                // We assume each ClassifierRole only has ONE state_machine!
                this.statemachines.put(context, state_machine);
            }
        }
    }
    return (Element[]) this.statemachines.values().toArray(
                                                    new Element[statemachines.size()]);
}

/**
 * Generates a 'list of pairs' of associated <code>Element</code> objects.
```

```
     * <p>
     * Each association in the Collaboration digram of the UML model associates two elements.
     * These associations are recorded in the <code>associations</code> 2d array.
     * The array is filled the first time this method is called.
     * Subsequent calls need only return the array itself.
     *
     * <p>
     * @return A 2-dimensional array of associated <code>Element</code> objects.
     */
    private Element[][] getAssociations() {
        if (this.associations == null) {
            Diagnostics.say ("PEPA_Extractor.getAssociations called when associations == null");
            ArrayList results = new ArrayList();
            String ASSOCIATION =
                "Behavioral_Elements.Collaborations.AssociationRole";
            String TYPE = "Foundation.Core.AssociationEnd.type";
            String ENDROLE =
                "Behavioral_Elements.Collaborations.AssociationEndRole";
            NodeList associations =
                this.dom_doc.getDocumentElement().getElementsByTagName(ASSOCIATION);

            Diagnostics.say ("PEPA_Extractor.associations has length " + associations.getLength());

            if (associations.getLength() == 0) {
                Diagnostics.warning("The PEPA Extractor found no associations in the collaboration diagram.");
                Diagnostics.diagnostic("This usually signals an error in the input.  Any PEPA output following may");
                Diagnostics.diagnostic("be incorrect or fail to parse.  Continuing to attempt to complete.");
            }

            for (int i = 0; i < associations.getLength(); i++) {
                Element association = (Element) associations.item(i);
                NodeList end_role = association.getElementsByTagName(ENDROLE);
                Element[] two = new Element[2];
                two[0] = this.getByRef((Element) end_role.item(0), TYPE);
                two[1] = this.getByRef((Element) end_role.item(1), TYPE);
                results.add(two);
            }
            this.associations =
                (Element[][]) results.toArray(new Element[results.size()][2]);
        }
        return this.associations;
    }

    /**
     * Locates the <code>State Element</code> in which a specified <code>StateMachine Element</code> should initialise.
     * <p>
     * The <code>StateMachine ELement</code> object should have in it a <code>State Element</code> which represents
     * an initial state in the UML State Diagram. This state should have a single transition to a 'real' state.
     * This real state is the one in which we are interested.
     * <p>
     * The location of such an element is only performed the first time this method is called.
     * Subsequent calls need only perform a lookup in the <code>initial</code> hashtable.
     *
     * <p>
     * @param sm a <code>StateMachine Element</code> object
     * @return the 'initial' <code>State Element</code> object
     */
    private Element getInitialState(Element sm) {
        if (!this.initial.containsKey(sm)) {
            Element[] states = this.getStates(sm);
            for (int i = 0; i < states.length; i++) {
                if (states[i].getTagName().endsWith(".Pseudostate")) {
                    Element kind = Extractor.getChild(states[i], ".kind");
                    if (kind.getAttribute("xmi.value").equals("initial")) {
                        Element outgoing =
                            this.getByRef(states[i], ".outgoing");
                        Element target = this.getByRef(outgoing, ".target");
                        this.initial.put(sm, target);
                    }
                }
            }
        }
        return (Element) this.initial.get(sm);
    }

    /**
     * Generates a list of the <code>State Element</code> objects present within a specified <code>StateMachine Element</code>.
     * <p>
     * This generation is only performed the first time this method is called with a particular <code>StateMachine Element</code> object.
     * Subsequent calls for the same <code>StateMachine Element</code> need only perform a lookup in the <code>states</code> hashtable.
     *
     * <p>
     * @param sm a <code>StateMachine Element</code> object
     * @return All the contained <code>State Element</code> objects
     *
     * @see PEPA_Extractor#getStates_helper
     */
    private Element[] getStates(Element sm) {
        if (!this.states.containsKey(sm)) {
            Element top = Extractor.getChild(sm, ".top");
            Element cs = Extractor.getChild(top, ".CompositeState");
            Element[] cse = PEPA_Extractor.getStates_helper(cs);
            if (cse == null) {
                Diagnostics.warning("converting null return to empty array");
                cse = new Element[0];
            }
            this.states.put(sm, cse);
        }
        return (Element[]) this.states.get(sm);
    }

    /**
     * Generates a list of the <code>State Element</code> objects present within a specified <code>CompositeState Element</code>.
     * <p>
     * This method can be called recursively.
     *
```

```
     * <p>
     * @param cs a <code>CompositeState Element</code> object
     * @return All the contained <code>State Element</code> objects
     */
    private static Element[] getStates_helper(Element cs) {
        Diagnostics.say ("Entered PEPA_Extractor.getStates_helper");
        ArrayList result = new ArrayList();
        Element sv = Extractor.getChild(cs, ".subvertex");
        if (sv == null) {
            Diagnostics.say("Extractor.getChild returned null, trying to cope ...");
            return null;
        }
        NodeList children = sv.getChildNodes();
        for (int i = 0; i < children.getLength(); i++) {
            Node child = children.item(i);
            if (child instanceof Element) {
                Element e = (Element) child;
                result.add(e);
                if (e.getTagName().endsWith(".CompositeState")) {
                    Element[] additional = PEPA_Extractor.getStates_helper(e);
                    for (int j = 0; j < additional.length; j++) {
                        result.add(additional[j]);
                    }
                }
            }
        }
        return (Element[]) result.toArray(new Element[result.size()]);
    }

    /**
     * Composes the full PEPA syntax output.
     *
     * <p>
     * @return An array of <code>String</code> objects representing the lines of PEPA
     *
     * @see PEPA_Extractor#getPEPA_Rates
     * @see PEPA_Extractor#getPEPA_definitions
     * @see PEPA_Extractor#getPEPA_cooperation
     */
    private String[] generatePEPA() {
        ArrayList results = new ArrayList();

        java.util.List defs = Arrays.asList(this.getPEPA_definitions());

        results.addAll(Arrays.asList(this.getPEPA_rates()));
        results.add("");
        results.addAll(defs);
        results.add("");
        results.add(this.getPEPA_cooperation());
        return (String[]) results.toArray(new String[results.size()]);
    }

    /**
     * Generates default values for all variable rates used in the model.
     *
     * <p>
     * @return An array of <code>String</code> objects representing the lines
     */
    private String[] getPEPA_rates() {
        String[] results = new String[this.rates.size()];

        for (int i = 0; i < this.rates.size(); i++)
            results[i] = "% " + (String) this.rates.get(i) + " = 2.0;";
        return results;
    }


    /**
     * Writes the PEPA to file, generating the filename from the original input file.
     *
     * <p>
     * @return the filename of the <code>.pepa</code> file
     */
    public String writePEPA() {

        String[] PEPA = this.generatePEPA();
        String name = this.xmi_file.getAbsolutePath();
        name = name.substring(0, name.lastIndexOf(".")) + ".pepa";

        try {
            FileWriter writer = new FileWriter(name);
            for (int i = 0; i < PEPA.length; i++)
                writer.write(PEPA[i] + "\n");
            writer.flush();
            writer.close();
        } catch (Exception e) {
            e.printStackTrace();
            return "";
        }
        return name;
    }
}
```

A.3 Coop.java

```
package pepa.extractor;

import java.util.HashSet;
import java.util.Arrays;
import org.w3c.dom.Element;



/**
 * The <code>Coop</code> interface generalises the nodes and leafs of the cooperation tree.
 */
interface Coop extends java.lang.Comparable {
    public String toString();
    public boolean contains(CoopLeaf leaf);
    public boolean contains(Element state);
}



/**
 * The <code>CoopNode</code> class represents a cooperation tree.
 */
class CoopNode implements Coop {

    /** The left branch - can be either a <code>CoopNode</code> or a <code>CoopLeaf</code> */
    public Coop left;

    /** An array of actions on which the left and right components should synchronise on */
    public String[] actions;

    /** The right branch - can be either a <code>CoopNode</code> or a <code>CoopLeaf</code> */
    public Coop right;

    /** Initialises a new <code>CoopNode</code> object with the specified components. */
    CoopNode(Coop left, String[] actions, Coop right) {
        this.left = left;
        this.actions = actions;
        this.right = right;
    }

    /** Compares two <code>CoopNode</code> objects. */
    public int compareTo(Object o) {
        CoopNode c = (CoopNode)o;
        return this.toString().compareTo(c.toString());
    }

    /**
     * Returns a PEPA formatted cooperation component.
     *
     * @see java.lang.Object#toString()
     */
    public String toString() {
        String left = this.left.toString();
        if (this.left instanceof CoopNode)
            left = "(" + left + ")";

        String actions = "";
        for (int i = 0; i < this.actions.length; i++) {
            if (i > 0)
                actions += ",";
            actions += this.actions[i];
        }
        actions = " <" + actions + "> ";

        String right = this.right.toString();
        if (this.right instanceof CoopNode)
            right = "(" + right + ")";

        return left + actions + right;
    }

    /** Returns true for an exact match in either branch */
    public boolean contains(CoopLeaf leaf) {
        return this.left.contains(leaf) || this.right.contains(leaf);
    }

    /** Returns true even if match is another instance in either branch */
    public boolean contains(Element state) {
        return this.left.contains(state) || this.right.contains(state);
    }

    /** Swaps left and right branches */
    public void twist() {
        Diagnostics.say("twisting this component");
        Coop temp = this.left;
        this.left = this.right;
        this.right = temp;
    }

    /** Springboard insert method - calls the 3-argument insert() function with true as third argument */
    public static CoopNode insert(Coop X, Coop Y) {
        return insert(X, Y, true);
    }

    /** Returns a new Node resulting from inserting Y into X. Recursive call tries inserting Y.twist(). */
    public static CoopNode insert(Coop X, Coop Y, boolean first) {
        if (X instanceof CoopLeaf && Y instanceof CoopLeaf) {
            return new CoopNode(X, new String[0], Y);
        } else if (X instanceof CoopLeaf) {
            return (CoopNode) Y;
        } else if (Y instanceof CoopLeaf) {
            CoopNode Old = (CoopNode) X;
            CoopLeaf n = (CoopLeaf) Y;
            if (Old.left.contains(n.state))
```

```
                return new CoopNode(
                                    insert(Old.left, n),
                                    Old.actions,
                                    Old.right);
            else if (Old.right.contains(n.state))
                return new CoopNode(
                                    Old.left,
                                    Old.actions,
                                    insert(Old.right, n));
            return Old;
        }

        CoopNode Old = (CoopNode) X;
        CoopNode New = (CoopNode) Y;
        CoopLeaf n_left = (CoopLeaf) New.left;
        CoopLeaf n_right = (CoopLeaf) New.right;

        if (Old.left.contains(n_left) && Old.right.contains(n_right)) {
            HashSet union = new HashSet(Arrays.asList(Old.actions));
            union.addAll(Arrays.asList(New.actions));
            String[] actions =
                (String[]) union.toArray(new String[union.size()]);
            return new CoopNode(Old.left, actions, Old.right);

        } else if (Old.left.contains(n_left)) {
            if (Old.right.contains(n_right.state)) {
                HashSet h = new HashSet(Arrays.asList(Old.actions));
                if (h.containsAll(Arrays.asList(New.actions)))
                    return new CoopNode(
                                        Old.left,
                                        Old.actions,
                                        insert(Old.right, n_right));
            }
            return new CoopNode(insert(Old.left, New), Old.actions, Old.right);

        } else if (Old.right.contains(n_right)) {
            if (Old.left.contains(n_left.state)) {
                HashSet h = new HashSet(Arrays.asList(Old.actions));
                if (h.containsAll(Arrays.asList(New.actions)))
                    return new CoopNode(
                                        insert(Old.left, n_left),
                                        Old.actions,
                                        Old.right);
            }
            return new CoopNode(Old.left, Old.actions, insert(Old.right, New));
        }

        if (first) {
            New.twist();
            return insert(Old, New, false);
        }
        return Old;
    }
}



/**
 * The <code>CoopLeaf</code> class represents a leaf of a cooperation tree.
 */
class CoopLeaf implements Coop {
    /** A <code>State Element</code> object */
    public Element state;

    /** A <code>ClassifierRole Element</code> object */
    public Element instance;

    /** Initialises a new <code>CoopLeaf</code> object. */
    CoopLeaf(Element state, Element instance) {
        this.state = state;
        this.instance = instance;
    }

    /** Compares two <code>CoopLeaf</code> objects. */
    public int compareTo(Object o) {
        CoopLeaf c = (CoopLeaf)o;
        return this.toString().compareTo(c.toString());
    }

    /** Returns true for an exact match */
    public boolean contains(CoopLeaf leaf) {
        return this.state == leaf.state && this.instance == leaf.instance;
    }

    /** Returns true even if match is another instance */
    public boolean contains(Element state) {
        return this.state == state;
    }

    /** Returns the name of the state of this component. */
    public String toString() {
        return PEPA_Extractor.getName(this.state);
    }
}
```

A.4 SyntaxStuff.java

```
package pepa.extractor;

import java.util.ArrayList;
import java.util.Comparator;

import pepa.process.Process;

/**
 * The <code>Definition</code> class represents the abstract syntax of a PEPA definition.
 * <p>
 * The classes of its components are found in the <code>pepa.process</code> package.
 * Ideally this class would feature in that package too. There may or may not already be a
 * class that fulfils the requirements implemented here.
 */
class Definition {

	/** The <code>Process.Var</code> that this definition is defining */
	private Process.Var v;

	/** The definition <code>Process</code> component */
	private Process p;

	Definition(Process v, Process p) {
		this.v = (Process.Var) v;
		this.p = p;
	}

	/** Returns valid PEPA syntax */
	public String toString() {
		return "# " + this.v.toString() + "\t= " + this.p.toString() + ";";
	}

	/** getter method */
	public Process.Var getVar() {
		return this.v;
	}

	/** getter method */
	public Process getProcess() {
		return this.p;
	}

	/** Puts a list of Definitions into the most logical ordering, given a starting point */
	public static Definition[] sort(Process.Var start, Definition[] list) {
		ArrayList sorted = new ArrayList();
		Definition current = getDefinition(start, list);
		sorted.add(current);

		while (sorted.size() < list.length) {
			ArrayList vars = getVars(current.getProcess());
			Process.Var[] others =
				(Process.Var[]) vars.toArray(new Process.Var[vars.size()]);
			for (int i = 0; i < others.length; i++) {
				Definition d = getDefinition(others[i], list);
				if (d != null && !sorted.contains(d))
					sorted.add(d);

			}
			current = (Definition) sorted.get(sorted.indexOf(current) + 1);
		}
		return (Definition[]) sorted.toArray(new Definition[sorted.size()]);
	}


	/** Returns from a list of Definitions, the Definition that matches the given Process.Var */
	private static Definition getDefinition(
		Process.Var var,
		Definition[] list) {
		for (int i = 0; i < list.length; i++) {
			if (list[i].getVar().equals(var))
				return list[i];
		}
		System.err.println("Looking for " + var.toString());
		System.err.flush();
		return null;
	}

	/** Returns a list of the Process.Var's present in a given Process */
	private static ArrayList getVars(Process p) {
		ArrayList result = new ArrayList();
		if (p instanceof Process.Var)
			result.add((Process.Var) p);
		else if (p instanceof Process.Prefix)
			result.addAll(getVars(((Process.Prefix) p).p));
		else if (p instanceof Process.Sum) {
			result.addAll(getVars(((Process.Sum) p).p1));
			result.addAll(getVars(((Process.Sum) p).p2));
		}
		return result;
	}
}

/**
 * The <code>PrefixComparator</code> class can be used to sort <code>pepa.process.Process.Prefix</code> objects.
 */
class PrefixComparator implements Comparator {

	/**
	 * Sorts two <code>Process.Prefix</code> objects based on the lexicographical ordering
	 * of their <code>Process.Var</code> components.
	 * <p>
	 * @see java.util.Comparator#compare(Object, Object)
	 */
	public int compare(Object o1, Object o2) {
```

```
                if (o1 instanceof Process.Prefix)
                        return compare(((Process.Prefix) o1).p, o2);
                else if (o2 instanceof Process.Prefix)
                        return compare(o1, ((Process.Prefix) o2).p);
                else if (o1 instanceof Process.Var && o2 instanceof Process.Var)
                        return 0 - Process.comp((Process) o1, (Process) o2);
                throw new ClassCastException();
        }
}
```

A.5 Diagnostics.java

```
package pepa.extractor;

/**
 * The <code>Diagnostics</code> class provides reporting routines for the Extractor.
 */
public class Diagnostics {

    /** A flag to indicate whether or not to run with verbose output. */
    public static boolean verbose = false;

    /** Called to print messages to the console, if verbose output selected. */
    public static void say(String s) {
        if (verbose) {
            System.out.println(s);
            System.out.flush();
        }
    }

    /** A counter to record the number of warnings printed. */
    public static int warnings = 0;

    /** Called to print messages to the console. */
    public static void warning(String s) {
        System.out.println("\nWarning>>> " + s);
        System.out.flush();
        warnings++;
    }

    /** Called to print messages to the console. */
    public static void diagnostic(String s) {
        System.out.println("Warning>>> " + s);
        System.out.flush();
    }

    /** Called to print messages to the console and exit. */
    public static void fatalError(String s) {
        System.out.println("Error>>> " + s);
        System.out.flush();
        System.exit(1);
    }


}
```

A.6 Reflector.java

```
package pepa.reflector;

import org.w3c.dom.*;
import java.io.*;
import java.util.Hashtable;
import java.util.zip.*;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;

/**
 * The <code>Reflector</code> class reflects results back to the original file.
 * <p>
 * The PEPA workbench supplied results in XML form. The state names in the original model are then
 * annotated with these results.
 */
public class Reflector {

        /** The number of decimal places required in the annotations */
        private static int DECIMAL_PLACES = 3;

        /** The original .xmi or .zargo file to reflect to. */
        private static File original;

        /** The .xml results file from the PEPA workbench. */
        private static File results;

        /** A flag to indicate whether or not to run with verbose output. */
        private static boolean verbose = false;

        /** Called by pepa.extractor.Extractor.parse(). */
        public static void setOriginal(File o) {
                original = o;
        }

        /** Called by pepa.pepa.Peperoni.solve(). */
        public static void setResults(File r) {
                results = r;
        }

        /** Called to print messages to the console. */
        private static void say(String s) {
            if (verbose) {
                System.out.println(s);
                System.out.flush();
            }
        }

        /**
         * Allow the reflector to be called from the command line.
         * <p>
         * This facility is used if solution is performed by some other PEPA tool, such as the
         * PRISM model checker.
         * <p>
         * @return <code>true</code> is reflection was successful, <code>false</code> otherwise
         */
        public static void main(String[] args) throws pepa.reflector.ReflectionException {
            String filename = "";

            if (args == null)
                throw new ReflectionException("No file name supplied");
            if (args.length == 2) {
                if (!args[0].equals("-verbose"))
                    throw new ReflectionException("Usage: java pepa.reflector.Reflector" +
                                                  " [-verbose] filename.{xmi|zargo}");
                verbose = true;
                filename = args[1];
            } else {
                if (args.length != 1)
                    throw new ReflectionException("Usage: java pepa.reflector.Reflector" +
                                                  " [-verbose] filename.{xmi|zargo}");
                filename = args[0];
            }

            say("Trying to load input UML file: " + filename);

            original = new File(filename);
            if (!original.exists())
                throw new ReflectionException("Could not find input UML file: " + filename);

            String path = filename.substring(0, filename.lastIndexOf("/") + 1);
            String basename = filename.substring(filename.lastIndexOf("/") + 1, filename.lastIndexOf("."));
            String resultsFile = path + "results/" + basename + ".xml";

            say("Trying to load results file: " + resultsFile);

            results = new File(resultsFile);
            if (!results.exists())
                throw new ReflectionException("Could not find results file: " + resultsFile);

            say("Starting reflection process");
            if (!reflect())
                throw new ReflectionException("Error processing file for reflection: " + filename);
            say("Finished reflection process, exiting successfully");
        }

        /**
         * Modify the state names of the model to include the PEPA annotations.
         * <p>
         * If the original filename was of the form <code>filename.{xmi|zargo}</code> then
         * the reflected model will have filename <code>filename.reflected.{xmi|zargo}</code>.
         * <p>
         * @return <code>true</code> is reflection was successful, <code>false</code> otherwise
         */
        public static boolean reflect() {
```

```
// If either file has not been set - not meant to be reflecting
if (original == null || results == null)
        return false;

// If either file is invalid - can't reflect
if (!original.exists() || !results.exists())
        return false;

say ("Building extractor");
pepa.extractor.Extractor extractor = new pepa.extractor.Extractor();

say ("Parsing input UML document");
extractor.parse(original);
Document o = extractor.getDocument();
say ("Parsing of input UML document complete");

say ("Parsing results UML document");
extractor.parse(results);
Document r = extractor.getDocument();
say ("Parsing of results UML document complete");

if (o == null || r == null)
        return false;

// The probability strings indexed by the state name
say ("Building state probability hash table");
Hashtable state_probability = new Hashtable();


say ("Populating state probability hash table");
// Fill the hashtable with the information from the XML results file.
NodeList states = r.getDocumentElement().getElementsByTagName("State");
for (int i = 0; i < states.getLength(); i++) {
        Node state = states.item(i);
        if (state instanceof Element) {
                Element e = (Element) state;
                String state_name = e.getAttribute("Name");

                say ("\tfound Element " + state_name);

                Element prob =
                        pepa.extractor.Extractor.getChild(e, "Probability");
                if (prob != null) {
                        String probability = prob.getFirstChild().getNodeValue();
                        say ("\t\tfound probability " + probability);
                        String value = "" + (Float.parseFloat(probability) * 100.0);
                        for (int j = 0; j < DECIMAL_PLACES; j++)
                            value += "0"; // pad with zeros
                        value =
                                value.substring(
                                        0,
                                        value.lastIndexOf(".") + 1 + DECIMAL_PLACES);
                        state_probability.put(state_name, "[" + value + "%]");
                }
        }
}


say ("Found all result probabilities");
say ("Starting reflection");

// Find each state and change its name to include the probability annotation.
NodeList all_states =
        o.getDocumentElement().getElementsByTagName(
                "Behavioral_Elements.State_Machines.State");
for (int i = 0; i < states.getLength(); i++) {
        Node state = all_states.item(i);
        if (state instanceof Element) {
                Element s = (Element) state;
                if (s.hasAttribute("xmi.id")) {
                        String name = pepa.extractor.Extractor.getName(s);
                        say ("\tFound name " + name);
                        if (state_probability.containsKey(name)) {
                                Element name_node =
                                        pepa.extractor.Extractor.getChild(s, ".name");
                                String new_string =
                                    name +  " " +
                                       (String) state_probability.get(name);
                                say ("\t\tBuilt new name " + new_string);
                                Text new_text = o.createTextNode(new_string);
                                // Replace the original name with the new annotated name
                                name_node.replaceChild(
                                        new_text,
                                        name_node.getFirstChild());
                                say ("\t\tReplaced new name " + new_string);
                        }
                }
        }
}

// If the file is an xmi file, just write it
if (original.getName().toLowerCase().endsWith(".xmi")) {
        String pathname = original.getAbsolutePath();
        String newpathname =
                pathname.substring(0, pathname.lastIndexOf("."))
                        + ".reflected.xmi";
        try {
                FileOutputStream out = new FileOutputStream( new File( newpathname ) );
                out.write( o.getDocumentElement().toString().getBytes() );
                out.flush();
                out.close();
        } catch (Exception e) {
                e.printStackTrace();
                return false;
        }
```

```
		} else if (original.getName().toLowerCase().endsWith(".zargo")) {
			// else we have to create the entire .zargo with the new .xmi file in it
			try {
				ZipInputStream in =
					new ZipInputStream(new FileInputStream(original));
				String pathname = original.getAbsolutePath();
				String newpathname =
					pathname.substring(0, pathname.lastIndexOf("."))
						+ ".reflected.zargo";
				ZipOutputStream out =
					new ZipOutputStream(
						new FileOutputStream(new File(newpathname)));

				ZipEntry entry = in.getNextEntry();
				while (entry != null) {
					out.putNextEntry(new ZipEntry(entry.getName()));

					// If we have reached the .xmi file in the .zargo
					if (entry.getName().toLowerCase().endsWith(".xmi")) {
						out.write(o.getDocumentElement().toString().getBytes());

					} else {
						// else just write the file out as normal.
						byte[] data = new byte[10240];
						while (in.available() != 0) {
							int read = in.read(data, 0, data.length);
							if (read == -1)
								break;
							out.write(data, 0, read);
						}
					}
					in.closeEntry();
					out.closeEntry();
					entry = in.getNextEntry();
				}
				in.close();
				out.finish();
				out.close();
			} catch (Exception e) {
				e.printStackTrace();
			}

		} else {
			return false;
		}
		return true;
	}
}
```