

# Documentation for Java Extractor / Reflector

Catherine Canevet, Matthew Prowse

September 11, 2002

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	Structure of This Document . . . . .	2
<b>2</b>	<b>What Does It Do?</b>	<b>3</b>
2.1	Extractor . . . . .	3
2.2	Reflector . . . . .	3
<b>3</b>	<b>How Does It Work?</b>	<b>4</b>
3.1	Extractor . . . . .	4
3.1.1	Component Definitions . . . . .	4
3.1.2	System Equation . . . . .	5
<b>4</b>	<b>How Is It Implemented?</b>	<b>10</b>
4.1	The <i>extractor</i> Package . . . . .	10
4.1.1	Extractor . . . . .	10
4.1.2	PEPA_Extractor . . . . .	10
4.2	The <i>reflector</i> Package . . . . .	12
<b>5</b>	<b>How It All Fits Together</b>	<b>13</b>
<b>6</b>	<b>What It Looks Like In Action</b>	<b>14</b>
<b>A</b>	<b>Example: Active Badge</b>	<b>16</b>
<b>B</b>	<b>Example: Server-Client</b>	<b>22</b>

# 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 modified `.xmi` or `.zargo` file is generated.

## 3 How Does It Work?

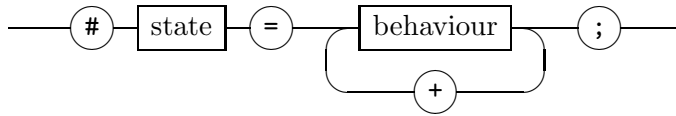
### 3.1 Extractor

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.

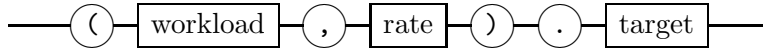
To generate a PEPA model, the Extractor extracts certain information from the UML model. The State Diagrams are used to generate *component definitions* and the Collaboration Diagram to generate the *system equation*.

#### 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*  $\leftarrow$  *empty list*
- 2: **for** each statemachine *S* **do**
- 3:   **for** each state *s* (of *S*) **do**
- 4:     *behaviors*  $\leftarrow$  *empty list*
- 5:     **for** each outgoing transition *t* (of *s*) **do**
- 6:       *w*  $\leftarrow$  name of trigger event of *t*
- 7:       *r*  $\leftarrow$  contents of “rate(...)” expression of *t*
- 8:       *tgt*  $\leftarrow$  name of target state of *t*
- 9:       *behaviors*  $\leftarrow$  *behaviors* + “(*w*, *r*).*tgt*”
- 10:    **end for**
- 11:    *n*  $\leftarrow$  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.

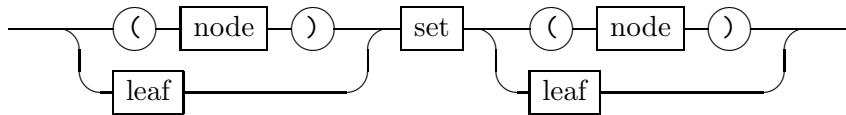
**Lines 6-9** generate the behaviour corresponding to the current transition. The event name and action expression of the transition, and the name of the target state define this behaviour.

**Lines 11-12** generate the term. The behaviours are separated by “+”.

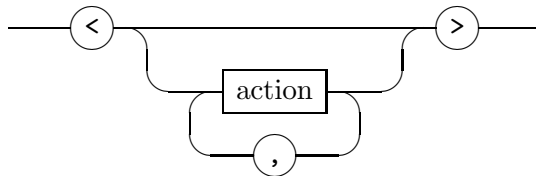
**Line 15** returns the generated terms.

### 3.1.2 System Equation

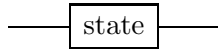
*node*



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

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

**Lines 2-11** comprise a *for* 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:  )
17: else if top.left contains new.left then
18:   if top.right contains another instance of new.right then
19:    if top.actions  $\cap$  new.actions  $\equiv$  new.actions then
20:      return Node( top.left, top.actions, insert( top.right, new.right
21:      ) )
22:    end if
23:  end if
24:  return Node( insert( top.left, new ), top.actions, top.right )
25: else if top.right contains new.right then
26:   if top.left contains another instance of new.left then
27:    if top.actions  $\cap$  new.actions  $\equiv$  new.actions then
28:      return Node( insert( top.left, new.left ), top.actions, top.right
29:      )
30:    end if
31:  end if
32:  return Node( top.left, top.actions, insert( top.right, new ) )
33: end if
34: if times > 0 then
35:   return top unchanged
36: end if
37: swap left and right leaves of new
38: 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 indefinitely. 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 returned. 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.

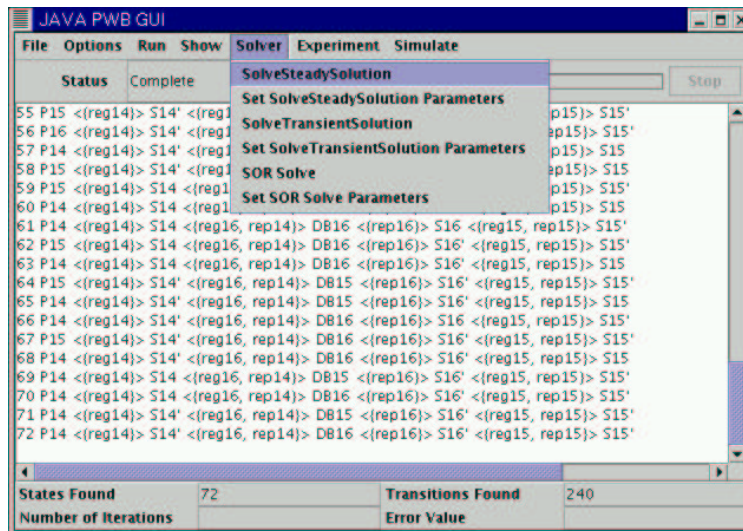
When the *PEPA\_Extractor* successfully parses a model, the *File* object is passed to the *Reflector* using its static *setOriginal()* method. Similarly, when the PEPA workbench solves a model and produces an *.xml* results file, a *File* object is passed to the *Reflector* using its static *setResults()* 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

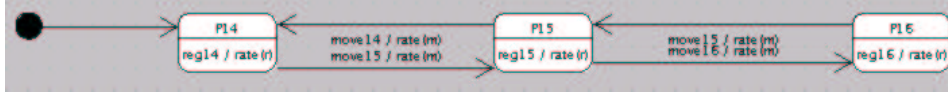


Figure 1: The Person state diagram

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;

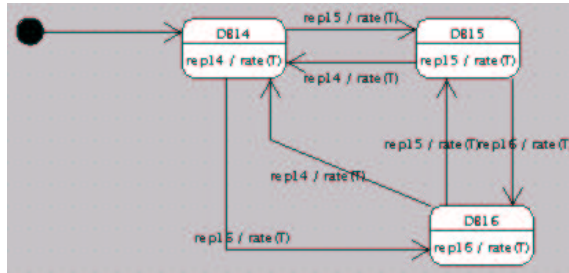


Figure 2: The Database state diagram

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;

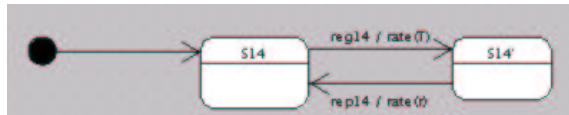


Figure 3: The Sensor S14 state diagram

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

```
# S14' = (rep14, s).S14;
```

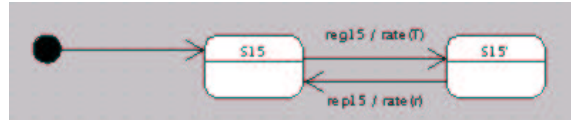


Figure 4: The Sensor S15 state diagram

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

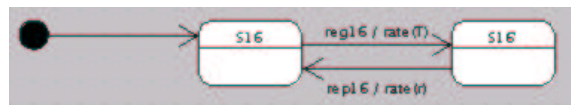


Figure 5: The Sensor S16 state diagram

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

## Generating the System Equation

Here is the collaboration diagram for this example :

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
```

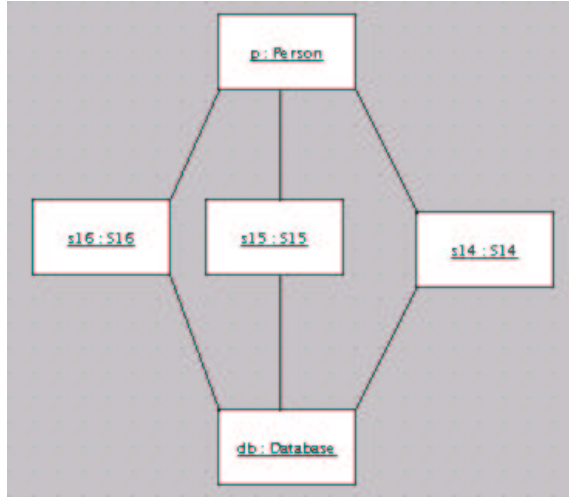


Figure 6: The collaboration diagram

```

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:

- |        |             |        |     |
|--------|-------------|--------|-----|
| $P14$  | $\boxtimes$ | $S15$  | (1) |
| $P14$  | $\boxtimes$ | $S16$  | (2) |
| $S15$  | $\boxtimes$ | $DB14$ | (3) |
| $S14$  | $\boxtimes$ | $DB14$ | (4) |
| $DB14$ | $\boxtimes$ | $S16$  | (5) |
| $P14$  | $\boxtimes$ | $S14$  | (6) |

**Declare** *top*

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

**P14**  $\boxtimes$  **S15**  
 $reg_{15}$

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)**

P14  $\bowtie_{reg16}$  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*:

(P14  $\bowtie_{reg16}$  S16)  $\bowtie_{reg15}$  S15

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

**Try to insert (3)**

S15  $\bowtie_{rep15}$  DB14

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

DB14  $\bowtie_{rep15}$  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*:

(P14  $\bowtie_{reg16}$  S16)  $\bowtie_{reg15}$  (DB14  $\bowtie_{rep15}$  S15)

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

**Try to insert (4)**

S14  $\bowtie_{rep14}$  DB14

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

DB14  $\bowtie_{rep15}$  S15

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

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

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*:

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

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

**Try to insert (5)**

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

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

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

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):

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

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

**Try to insert (6)**

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

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):

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

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

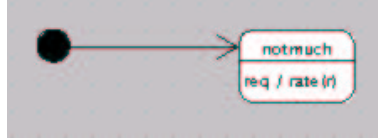


Figure 7: The User state diagram

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

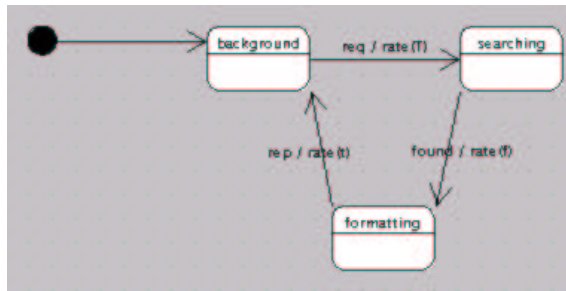


Figure 8: The Server state diagram

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

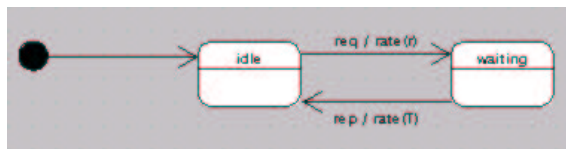


Figure 9: The Client state diagram

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 :

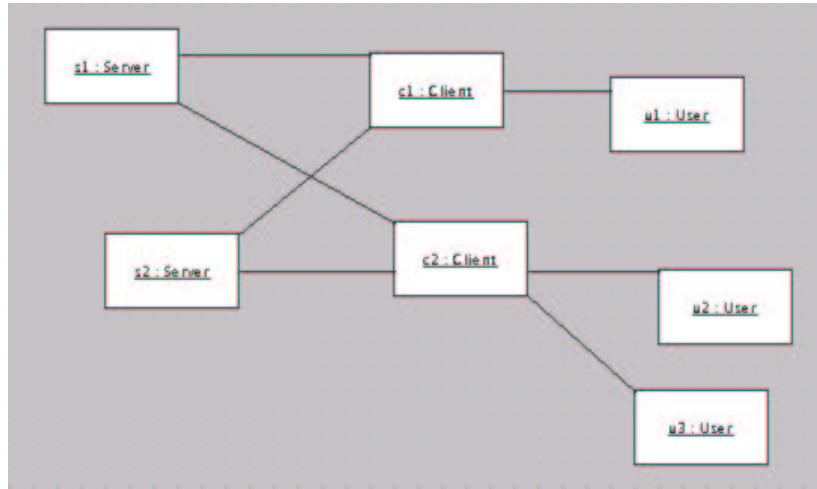


Figure 10: The collaboration diagram

The atomic cooperations are the following :

$$S1 \quad \begin{array}{c} \boxtimes \\ req,rep \end{array} \quad C1 \quad (7)$$

$$S2 \quad \begin{array}{c} \boxtimes \\ req,rep \end{array} \quad C2 \quad (8)$$

$$S1 \quad \begin{array}{c} \boxtimes \\ req,rep \end{array} \quad C2 \quad (9)$$

$$U3 \quad \begin{array}{c} \boxtimes \\ req \end{array} \quad C2 \quad (10)$$

$$U1 \quad \begin{array}{c} \boxtimes \\ req \end{array} \quad C1 \quad (11)$$

$$S2 \quad \begin{array}{c} \boxtimes \\ req,rep \end{array} \quad C1 \quad (12)$$

$$U2 \quad \begin{array}{c} \boxtimes \\ req \end{array} \quad C2 \quad (13)$$

### Step 1

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

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

**Step 2 : Try to insert (2)**

$$\begin{aligned} top &: \mathbf{S1} \underset{req,rep}{\boxtimes} \mathbf{C1} \\ new &: \mathbf{S2} \underset{req,rep}{\boxtimes} \mathbf{C2} \end{aligned}$$

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)**

$$\begin{aligned} top &: \mathbf{S1} \underset{req,rep}{\boxtimes} \mathbf{C1} \\ new &: \mathbf{S1} \underset{req,rep}{\boxtimes} \mathbf{C2} \end{aligned}$$

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) )

$$\begin{aligned} top &: \mathbf{C1} \\ new &: \mathbf{C2} \end{aligned}$$

They are both instances so we have :  $\mathbf{C1} \boxtimes \mathbf{C2}$

$$\text{And : } \mathbf{S1} \underset{req,rep}{\boxtimes} (\mathbf{C1} \boxtimes \mathbf{C2})$$

**Step 4 : Try to insert (4)**

$$\begin{aligned} top &: \mathbf{S1} \underset{req,rep}{\boxtimes} (\mathbf{C1} \boxtimes \mathbf{C2}) \\ new &: \mathbf{U3} \underset{req}{\boxtimes} \mathbf{C2} \end{aligned}$$

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))

$$\begin{aligned} top &: \mathbf{C1} \boxtimes \mathbf{C2} \\ new &: \mathbf{U3} \underset{req}{\boxtimes} \mathbf{C2} \end{aligned}$$

C2 is in both right parts. The left parts do not match. So it returns Node(top.left, top.actions, insert(top.right, new)).

$$top : \mathbf{C2}$$

$new : U3 \bowtie_{req} C2$

Top is a leaf so it returns new.

And :  $S1 \bowtie_{req,rep} (C1 \bowtie_{req} (U3 \bowtie_{req} C2))$

**Step 5 : Try to insert (5)**

$top : S1 \bowtie_{req,rep} (C1 \bowtie_{req} (U3 \bowtie_{req} C2))$

$new : U1 \bowtie_{req} C1$

We have C1 in both right parts. Left parts do not match. It returns Node (top.left, top.actions, insert(top.right, new) )

$top : C1 \bowtie_{req} (U3 \bowtie_{req} C2)$

$new : U1 \bowtie_{req} C1$

No matches, it returns unable. It twists the new one which becomes : C1

$\bowtie_{req} U1$

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).

$top : C1$

$new : C1 \bowtie_{req} U1$

Top is a leaf so it returns new.

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

**Step 6 : Try to insert (6)**

$top : S1 \bowtie_{req,rep} ( (C1 \bowtie_{req} U1) \bowtie_{req} (U3 \bowtie_{req} C2) )$

$new : S2 \bowtie_{req,rep} C1$

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

### 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 ) )
```