Stochastic models for quality of service of component connectors

Moon, Y.J.

Chapter 5

Tool implementation

5.1 Introduction

The growing complexity and importance of coordination models in software applications necessarily lead to a higher relevance of performance issues for coordinators in the development of systems. In this context, the performance of such models plays an important role in the quality of the final software system. Unfortunately, the lack of tools that support the performance analysis of coordination models makes it difficult to automatically analyze the performance properties of coordination models.

In this chapter, we introduce a tool that integrates a Stochastic Reo editor and a generator of CTMCs from Stochastic Reo models. The Stochastic Reo editor is an extension of the existing Reo editor in the Extensible Coordination Tools (ECT) [35], which is an integrated toolset for the design and the verification of (Stochastic) Reo. We have implemented the CTMC generator based on the semantics of Quantitative Intentional Automata (QIA) and their translation algorithm presented in Chapter 3. This tool, called *Reo2MC*, is provided as a plug-in for the ECT.

This chapter consists of two parts: (1) the description of the implementation of the Reo2MC tool and (2) a manual for its usage. In the first part, we explain the data structures for certain elements such as stochastic rates and delay-sequences, which were mentioned in the previous chapters.

In the second part, we explain how to use the tool: for instance, how to generate a QIA model corresponding to a Stochastic Reo connector, or translate a CTMC model from a Stochastic Reo connector or a QIA model generated from a Stochastic Reo connector. In addition, we show the link to other stochastic analysis tools, in particular PRISM [76, 57].

5.2 Reo2MC: description and implementation

Reo2MC is a plug-in for the ECT, which was introduced in [8]. The tool allows users to draw Stochastic Reo models, using an extension of the existing Reo editor

in ECT, and automatically derive the QIA semantics of Stochastic Reo models and their corresponding CTMCs.

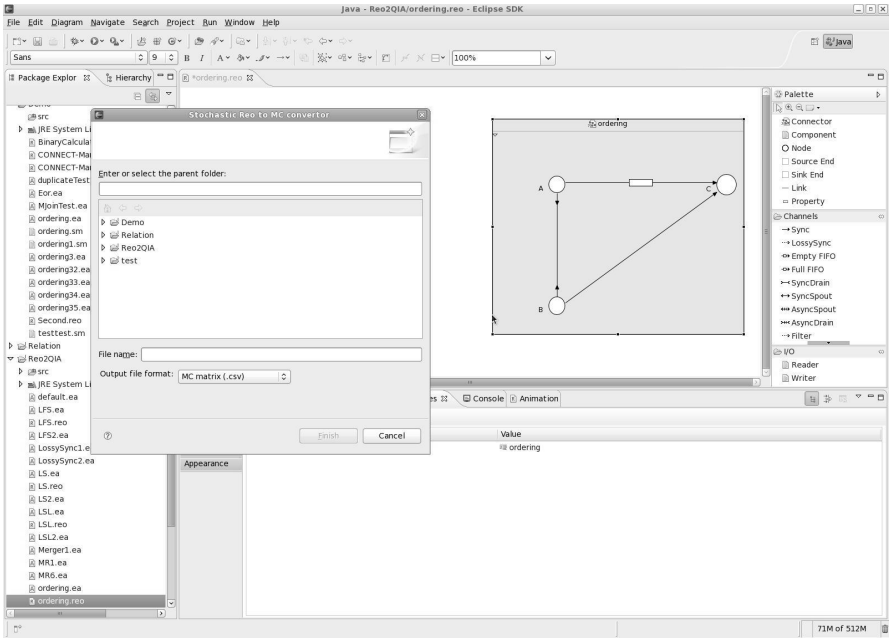


Figure 5.1: A Snapshot of Reo2MC

Reo2MC works as an Eclipse application through a graphical user interface (see Figure 5.1). The execution flow is depicted in Figure 5.2: the user provides as input a Reo circuit (which is obtained either by using the graphical editor in ECT, or by automatic synthesis from other specifications, such as UML sequence diagrams or BPMN models, the tools for which are also provided in ECT [28]) and a textual description of the stochastic constraints on the connector and its environment. Once all this input has been provided, the model can be automatically translated into QIA and CTMCs, both represented in XML. The GMF framework in Eclipse is used to generate the graphical representations of QIA and CTMCs, and the XML files of the generated CTMCs can be parsed into several other file formats, which are used as input to PRISM, MATLAB, and Maple to analyze the performance of connectors.

5.2.1 Implementation

In this section, we show the data structures and underlying functionalities used for the implementation of Reo2MC. In Reo2MC, we have implemented QIA as an operational semantic model for Stochastic Reo and as an intermediate model for the translation to CTMCs.

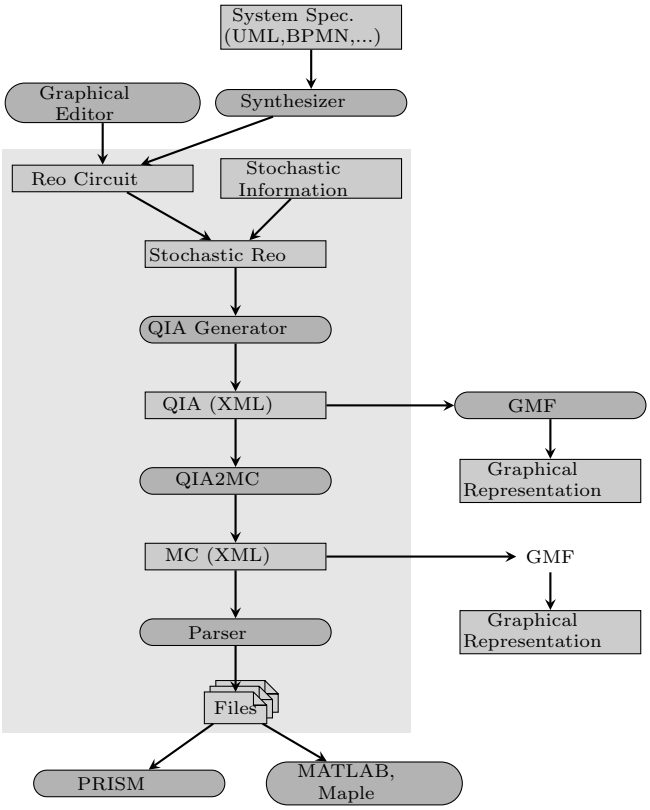


Figure 5.2: Architecture of Reo2MC

The main data structures, the implementations of which are explained in this section, consist of *structural stochastic information* for the activities involved in Reo channels and a *delay-sequence* which is the sequence of individual structural stochastic information for synchronized activities. The implementation of these data structures is based on their definitions and features as covered in Chapters 3 and 4.

The functionalities implemented in Reo2MC are also based on the definitions and the algorithms in Chapters 3 and 4. The main functionalities of Reo2MC are the product of QIA, and the extraction and the division of delay-sequences to generate their corresponding CTMCs from QIA. The following sections show the implementation details of these functionalities.

Data structures

Structural stochastic rates In Stochastic Reo, stochastic values are used to represent the arrival rates at each channel end and the processing delay rates of each

channel. The stochastic values must be non-negative real values since our target model is CTMCs, and hence are defined with type *double* in the implementation. For simplicity of modeling, we assume that each stochastic process has only one rate value.

In the existing Reo editor, a primitive Reo channel consists of two types of objects: two channel ends and an arrow between the two channel ends. A channel end is associated with one *double* value (its arrival rate), whereas the arrow in a channel is able to have more than one *double* value, according to its behavior. For instance, a Sync has one processing delay rate for a data-flow from its source to its sink nodes; a FIFO1 has two processing delay rates for data-flows from its source node to its buffer and from its buffer to its sink node; a LossySync also has two processing delay rates for a successful data-flow from its source to its sink nodes and data-loss at its source node.

Thus, we define the data type of the rates as an array of *double* values. According to the assumptions mentioned above, an arrival rate of I/O request is only one value, which we assume to be the first value in the array, i.e., the rest will be ignored. In the case of processing delay rates, the values in the array are sequentially taken as corresponding to the actions of a channel from a successful processing (e.g., a data-flow in a LossySync) to a non-successful processing (e.g., a data-loss in a LossySync) or its sequential data-flows (e.g., in a FIFO1 data-flow from its source node to the buffer and from the buffer to its sink node correspond to, respectively, the first and the second values in the array). This *double* array can be a general data structure, but this array itself is not user-friendly because it requires users to have the insight of this array, e.g., the meaning of the order of elements in the array. The implementation in the Reo editor provides some guidelines to users by presenting specific rate labels for respective Reo channels. The way of setting rates is shown in Section 5.2.2.

These values are propagated to and reused in a QIA model, corresponding to a given Stochastic Reo connector, as the elements in the labels on QIA transitions. The mapping from Stochastic Reo to QIA is carried out as follows. First, primitive channels comprising a connector are mapped to their corresponding QIA models. Since mapping the primitive channels to their corresponding QIA is a one-to-one mapping, this is quite straightforward. Then the product of the corresponding QIA models generates the QIA model for the connector. Thus, the actual pairing of a rate with its relevant activity takes place during the mapping of primitive channels to their corresponding QIA.

This pairing is decided according to the node names relevant to each stochastic process, as the rates are named after their respective node names in Stochastic Reo (this naming is explained on page 11). In addition, for the translation to CTMC, the rates need to be delineated according to the connection information of the Stochastic Reo connector. For this purpose, we propose structural stochastic information that describes the stochastic rates with explicit mention of their relevant source (input) and sink (output) nodes. Such structural stochastic information is defined as a 3-tuple and represented as an element in the labels of QIA transitions. Such a tuple, denoted by a `DelayElement`, has been implemented as:

```
public class DelayElement extends EObjectImpl implements EObject {
  protected EList<String> input;
  protected EList<String> output;
  protected double delay = DELAY_EDEFAULT;
  ...
}
```

The attributes in this class are `input` (source nodes), `output` (sink nodes), and `delay` (a rate).

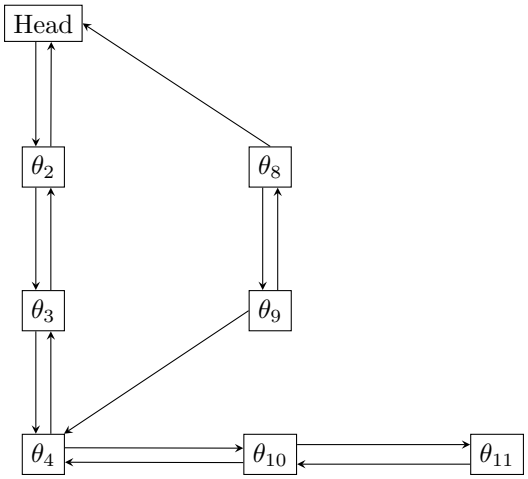
Delay-sequences In the translation from QIA to CTMC models, a delay-sequence, defined in Section 3.3.1, is generated for each transition of synchronized data-flows. Each data-flow has a 3-tuple $\theta = (I, O, r)$ that depicts its connection information reflecting the topology of a Reo connector, which is implemented by `DelayElement`.

A delay-sequence is composed by the operators | and ; for, respectively, parallel and sequential compositions. A delay-sequence composed by | describes that the data-flows corresponding to each element in the delay-sequence occur interleaved. A delay-sequence composed by ; describes that the data-flows occur sequentially, from the leftmost element to the rightmost one.

In order to represent such delay-sequences, we define our own linked-list and its elements, called `DElmNode`. The content of the elements of this linked-list is the tuple θ, i.e., the data type E below is `DelayElement`. Each `DElmNode` has four pointers of `prev`, `next`, `right`, and `left` as shown below.

```
public class DElmNode<E> {
   public DElmNode<E> prev = null;
   public DElmNode<E> next = null;
   public DElmNode<E> left = null;
   public DElmode<E> right = null;
   public E data = null;
   ...
}
```

We use the above pointers to generate a linked-list, representing a delay-sequence. The following example shows the concrete idea of the structure of our linked-list. Recall the delay-sequence $\lambda_1 = ((\theta_2; \theta_3)|(\theta_8; \theta_9)); (\theta_4|\theta_{10}|\theta_{11})$ in Example 3.3.1. The linked-list representation of this delay-sequence is described as:



where we have omitted redundant pointers that indicate *null*.

In the implementation of our linked-list (see below), there are two different ways to insert nodes: horizontal and vertical insertion. At a current node, horizontal inserting is used to insert a node that is composed with the current node in parallel; vertical inserting is used to insert a node that is composed with the current node sequentially. In implementation, horizontal and vertical insertings are described by the functions `parallelInsertion` and `sequentialInsertion`, respectively.

```
public class NewLinkedList{
   DElmNode<DelayElement> head = null; //The head of the list
   DElmNode<DelayElement> tail = null; //The tail of the list
   DElmNode<DelayElement> current = null; //Last modified node
                                          //of the list
   ...
   public void sequentialInsertion(DelayElement f){
      if(this.isEmpty())    addDatatoTail(f);
      else if(this.current.next==null){
         DElmNode<DelayElement> temp = new DElmNode<DelayElement>();
         temp.prev = current;
         temp.next = null;
         temp.left = null;
         temp.right = null;
         temp.data = f;
         current.next = temp;
         current = temp;
         tail = temp;
     }
   }
```

```
   public void parallelInsertion(DelayElement b){
      if(!this.isEmpty() && current!=null){
         DElmNode<DelayElement> temp = new DElmNode<DelayElement>();
         temp.prev = current.prev;
         temp.next = current.next;
         temp.left = current.getLastright();
         temp.left.right = temp;
         temp.right = null;
         temp.data = b;
      }
   }
   ...
}
```

Algorithms

Next, we explain the implementation of the algorithms for the mapping between stochastic Reo and QIA, the product operation of QIA, and the translating of a QIA to a CTMC.

Obtaining QIA for Stochastic Reo We obtain the QIA model corresponding to a Reo connector in two steps:

1. map primitive channels, which constitute the connector, to their corresponding QIA models.
2. compose the QIA models obtained from Step 1.

We assume that the types of primitive Reo channels are fixed (Sync, LossySync, SyncDrain, FIFO1, and so on). Thus, we use a template that provides a one-to-one mapping for each primitive channel and its QIA models.

Each QIA model in the template uses temporary names for its nodes such as `SOURCE0` and `SINK0`. These temporary names in a QIA model are renamed according to the node names of the primitive channel corresponding to the QIA model. When a node is shared by more than two channels, e.g., a replicator or a merger, indices are used in the renaming procedure to explicitly describe the processing delay rates of the channels. The following code shows the implementation of this renaming where `sourceName` is a name given by users and `SOURCE` is a default name in the QIA template:

```
int i = 0;
for (PrimitiveEnd sourceEnd: channel.getSourceEnds()){
  String sourceName = endNames.getName(sourceEnd);
  QIARefactoring.renamePortName(SOURCE+ i++, sourceName, copy);
  ...
}
```

`SOURCE` is replaced with `sourceName`, and the index `i` is attached to the replaced name.

Like temporary node names in the template, the values of processing delay rates also use null temporary values. During the mapping, these values will be updated according to the user's input. However, providing values is optional at this stage and can be postponed until the CTMC model corresponding to a connector is generated. Next, we describe how to compose QIA models, by means of the product operation.

QIA product According to the QIA product in Definition 3.2.2, the resulting transition of the product of a pair of transitions fires them together, in case they agree on the common nodes of the two automata, or independently, otherwise. In the case of two synchronized firing transitions, the product result is implemented as joining all the elements of the two transitions. For instance, given two transitions $p_1 \xrightarrow{A|B,\Theta_1} p_2$ and $q_1 \xrightarrow{C|D,\Theta_2} q_2$, the composition result is $\langle p_1, q_1\rangle \xrightarrow{A\cup C|B\cup D,\Theta_1\cup\Theta_2} \langle p_2, q_2\rangle$. For the product of interleaved transitions, a different implementation is required, but for code reusability, we decided to reuse this joining method. For this purpose, a null transition $q \xrightarrow{\emptyset|\emptyset,\emptyset} q$ is added to all states in the two automata as pre-processing for the product. Then, for any interleaved transition $p_1 \xrightarrow{A|B,\Theta_1} p_2$, its composition result is $\langle p_1, q\rangle \xrightarrow{A|B,\Theta_1} \langle p_2, q\rangle$.

Extracting delay-sequences The extraction of a delay-sequence is implemented based on Algorithm 3.3.1. According to this algorithm, each independent sub-delay-sequence λ_θ is generated and then the sub-delay-sequences are composed in parallel to generate the whole delay-sequence S. Each initial 3-tuple itself becomes a starting point for an independent sub-delay-sequence. As a newly appended 3-tuple, each initial one is used to choose adjacent 3-tuples, which are appended to the end of the relevant sub-delay-sequences until no more adjacent 3-tuples exist. In the following implementation, `Post` is used to denote the set of adjacent 3-tuples and it is appended to the linked-list mentioned in Section 5.2.1, which is the data structure for a delay-sequence. This function is a direct implementation of Algorithm 3.3.1.

```
List<DelayElement> Init = getEdgeInput(DI);
List<NewLinkedList> dlist = new Vector<NewLinkedList>();

for(DelayElement a : Init){
  NewLinkedList temp = new NewLinkedList();
  temp.sequentialInsertion(a);
  dlist.add(temp);

  List<DelayElement> Pre = new Vector<DelayElement>();
  List<DelayElement> Post = new Vector<DelayElement>();
  Pre.add(a);
```

```
  Post = getNext(DI,Pre);
  while(!Post.isEmpty()){
    for(DelayElement b : Post){
      temp.contains_removes(b);
    }
    temp.sequentialInsertion(Post.get(0));
    for(int i=1;i<Post.size();i++){
      temp.parallelInsertion(Post.get(i));
    }
    Pre.clear();
    Pre.addAll(Post);
    Post = getNext(DI,Pre);
  }
}
```

Deriving CTMC We discuss how to divide a macro-step transition with a delay-sequence into a number of micro-step transitions. This is implemented based on the function *div*, which is explained in Section 3.3.3. The first and the fourth conditions in the *div* function are trivial, thus, here, we consider the second and the third conditions only.

In the implementation, an element `Current` (see below) in a linked-list, which points to the current position of the list, traverses the linked-list corresponding to a delay-sequence. We extract the necessary information to generate a CTMC model from the structural properties of the nodes in this list. For example, when the `Current` element is a *head* node of a list, it corresponds an initial state of a CTMC model to be generated. When the `Current` element is an actual element, this element must be handled by adding a new transition with a new target state to the CTMC state that is generated by its `prev` element. Moreover, when the pointer `right` of the `Current` element indicates an actual element, the current element and its `right` element must be interleaved.

We now consider the second and third conditions in the *div* function. The second condition in *div* implies a sequentially composed delay-sequence. In this case, `right` of the `Current` element is *null*. While traversing a linked-list, e.g., `Current = Current.next`, it generates a linear state diagram corresponding to the sequentially composed delay-sequence represented by the list.

```
while(Current!=null){
  ...
  else if(Current.right==null){
    State source = preTarget;
    State target = new State();
    Transition t = new Transition();
    ...
    t.setSource(source);
```

```
    t.setTarget(target);
    Current = Current.next;
  }
  ...
}
```

The third condition in *div* implies a delay-sequence composed in parallel. Then `right` of the `Current` element must not be *null*. We divide this into two different cases according to their possible structure: first a number of delay-sequences are composed in parallel; second a number of 3-tuples are composed in parallel. These two cases are denoted by, respectively, `parallelList` and `parallelNode` in the implementation, and distinguished by the types of the `Current` element, i.e., a *head* or a normal node. For example, if the `right` of the `Current` element is a *head* node, then it implies that a delay-sequence is composed in parallel since the instance of each delay-sequence is a linked-list and every linked-list starts with a *head* node. Thus, the `parallelList` case is considered to generated a CTMC model.

```
if(Current.right!=null){
  DElmNode<DelayElement> horizontal = Current;
  parallelList = false;
  parallelNode = false;
  while(horizontal!=null){
    if(horizontal.isHead()){
      horizontal.setVisited(true);
      store.push(horizontal.next);
      parallelList = true;
    }
    else{
      store.push(horizontal);
      parallelNode = true;
    }
    horizontal = horizontal.right;
  }
  ...
}
```

The elements connected to the `Current` element in the linked-list by the pointers `left` and `right` are stored in the stack `store` in the implementation (see below). The stored elements are used to generate the corresponding CTMC fragment according to whether their structure identifies them as *parallelList* or *parallelNode*.

The CTMC fragment for *parallelList* is built as follows. Let L be a linked-list and $l_1, \ldots, l_n$ be the linked-lists that are composed in parallel to constitute L. We first generate the CTMC fragments corresponding to $l_1, \ldots, l_n$, in a recursive procedure, since each l_i is also a linked-list. After that, the CTMC fragments of all l_i are interleaved. Thus, the CTMC fragment for the whole linked-list L guarantees the independence of $l_1, \ldots, l_n$ while retaining the precedence order of the elements in each l_i.

```
while(!store.isEmpty()){
  if(parallelList){
    DElmNode<DelayElement> tempNode = store.pop();
    DelayElement temp = tempNode.getData();
    ...
    Automaton tempAutomaton1 = new Automaton();
    NewLinkedList sub1 = list.subSeq4(temp);
    costA =
     product(costA, addNewParts(tempAutomaton1, transition, sub1));
  }
  ...
}
```

For a `parallelNode`, each element in the stack `store` corresponds to an automaton with two states and one transition, which is represented as `tempAutomaton2` below. The automata generated for these elements are also interleaved in the composition.

```
else if(parallelNode){
  while(!store.isEmpty()){
    DElmNode<DelayElement> tempElement2 = store.pop();
    tempElement2.setVisited(true);

    // Make an automaton for each parallel delay
    Transition tempTransition2 = new Transition();
    State tempSource2 = new State();
    State tempTarget2 = new State();
    tempTransition2.setAutomaton(tempAutomaton2);
    tempTransition2.setSource(tempSource2);
    tempTransition2.setTarget(tempTarget2);
    ...
    // Get an automaton of whole parallel delays
    costA = product(costA, tempAutomaton2);
  }
}
```

5.2.2 Usage

In this section, we explain the usage of Reo2MC. Reo2MC is a plug-in for the ECT, which generates the semantic model, QIA, of a Stochastic Reo circuit and translates it into its corresponding stochastic model, CTMCs. Depending on the type of analysis to be performed on a Stochastic Reo circuit, users can choose the target model of the translation as QIA or CTMCs.

The Stochastic Reo circuit input to Reo2MC provides auxiliary information such as explicit node names and stochastic rates for request-arrivals and data-flows. Node names are used to denominate rates that are used for the analysis. Thus, we need to

name only the nodes relevant to our stochastic processes of interest, instead of all the nodes in a circuit. The rate values do not necessarily have to be set in the drawing phase of the Reo circuits; their assignments can be postponed.

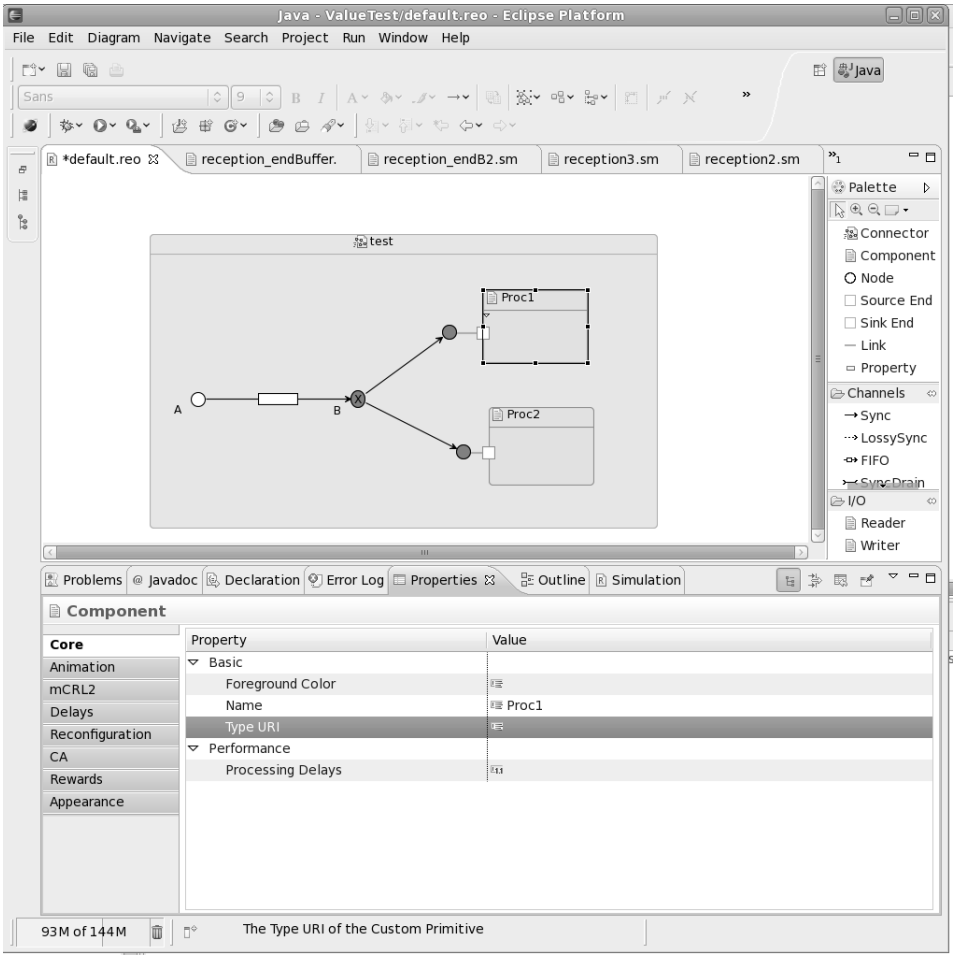


Figure 5.3: Hierarchical modeling in Reo editor

Reo editor

In the Reo editor[1], components and connectors between the components can be specified. A component in the Reo editor is described as a black box with source (input)

[1]The graphical user interface of Reo2MC, including Reo editor and the basic template for automata, have been implemented by Christian Krause. Details can be focused in his thesis [55].

and sink (output) ends to be connected to connectors. Such components function as the environment for connectors. Drawing a component is done intuitively as follows:

- select `Component` in the `Palette` at the left of the Reo editor (see Figure 5.3 which is a screen shot of the Reo editor.)
- click any spot at the editing canvas

As mentioned before, connectors are specified by composing basic Reo channels. The types of the basic channels are finite such as Sync, LossySync, and FIFO1. The Reo editor provides a template of these basic channels in the `Channels` section below the `Palette` section. Drawing connectors is very similar to the way of drawing components. That is, first draw a `Connector` at the editing canvas instead of a `Component`, then nodes and channels can be drawn inside the `Connector`. For instance, in Figure 5.3, a FIFO1 channel and two Sync channels are drawn inside the `Connector test`.

In addition, a component can be used hierarchically as a sub-component in connectors or other components. This provides better visualization and readability for modeling, and moreover, it guarantees reusability. See the model in Figure 5.3. The `Components` `Proc1` and `Proc2` belong to `Connector test`. The specification of these `Components` can be provided by external files that must include QIA or CA models corresponding to their behavior. The location of these external files are set in the `TypeURL` entry of the `Basic` field in the `Core` section in the `Properties` tab at the

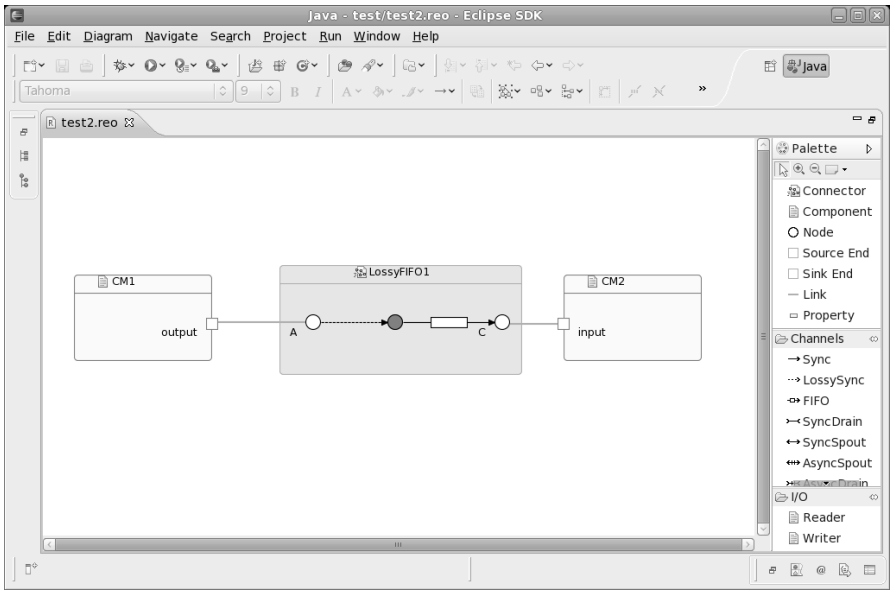


Figure 5.4: Connector between components

bottom of the Reo editor. In Figure 5.3, this `TypeURL` entry is selected for setting the location of external files. Such usage of external files for modeling enables one to reuse existing automata models for other specifications.

As the environment of connectors, components interact with their adjacent connectors through the boundary nodes of the connectors. Figure 5.4 shows a reader (CM2) and a writer (CM1) components and a LossyFIFO1 circuit that connects them in the Reo editor. Note that in Reo2MC, node names are represented in upper case. Even though one can input node names in lower case, Reo2MC changes them into upper case. Thus, here and in the remainder of this chapter, examples of Reo circuits have node names in upper case.

As mentioned before, a Stochastic Reo connector has two different kinds of stochastic values: arrival rates and processing delay rates. For setting these values, users first have to choose a node or a primitive channel in the editor by clicking on it. Then

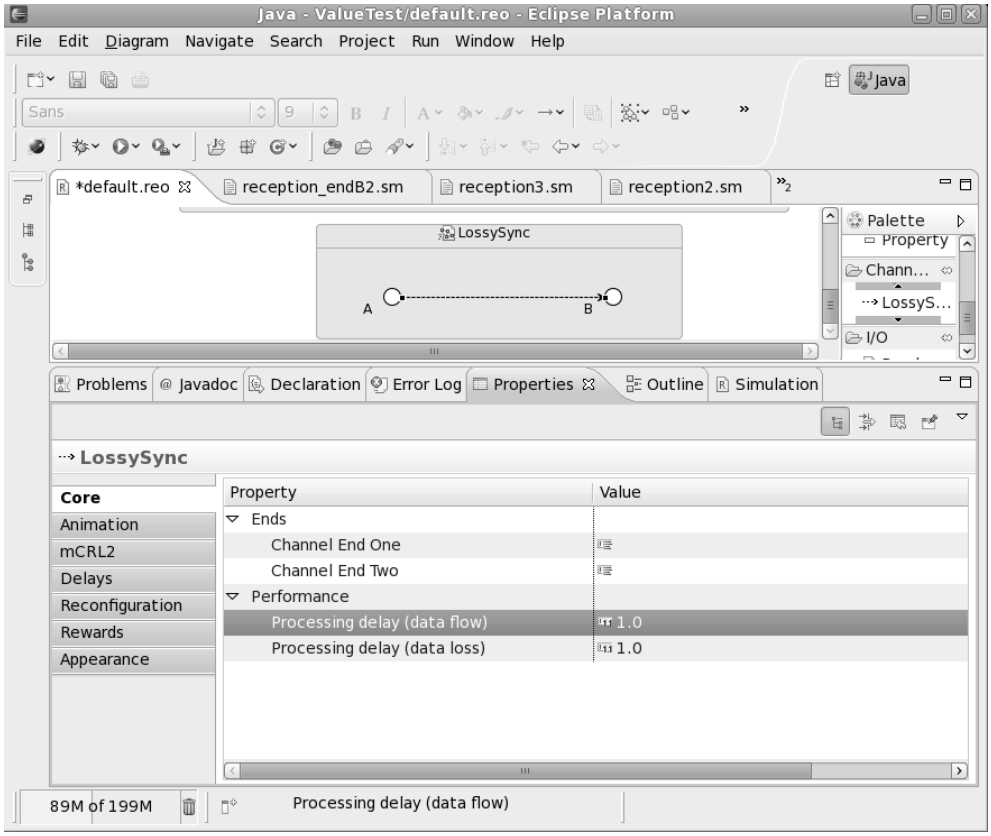


Figure 5.5: Setting processing delay rates for LossySync

the rates can be set in the `Performance` field of the `Core` section in the `Properties` tab at the bottom of the editor. As an example, Figure 5.5 shows how to set the processing delay rates of a LossySync circuit. According to the types of rates, i.e., arrival rates and processing delay rates, the `Performance` field shows relevant labels. In Figure 5.5, a channel is selected, thus, the `Performance` field shows the label `Processing delay`, otherwise, `Arrival rate` would be presented. Moreover, if a channel has more than one activity, e.g., a LossySync or a FIFO1, the `Performance` field presents all possible types of processing delay rates explicitly. For instance, for a LossySync channel, `Processing delay (data flow)` and `Processing delay (data loss)` in Figure 5.5.

Setting values for the rates is optional. For fixed rate values, users can set the values in this specification phase. However, if users want to analyze how the system's behavior changes by tweaking the rates, then the rates can be left without setting their values to be decided after the translation into a CTMC model.

In order to set and manipulate the rate values in a derived stochastic model, users need to fix the node name before the translation procedure. Otherwise, Reo2MC will decide node names automatically and this makes it difficult to figure out which name

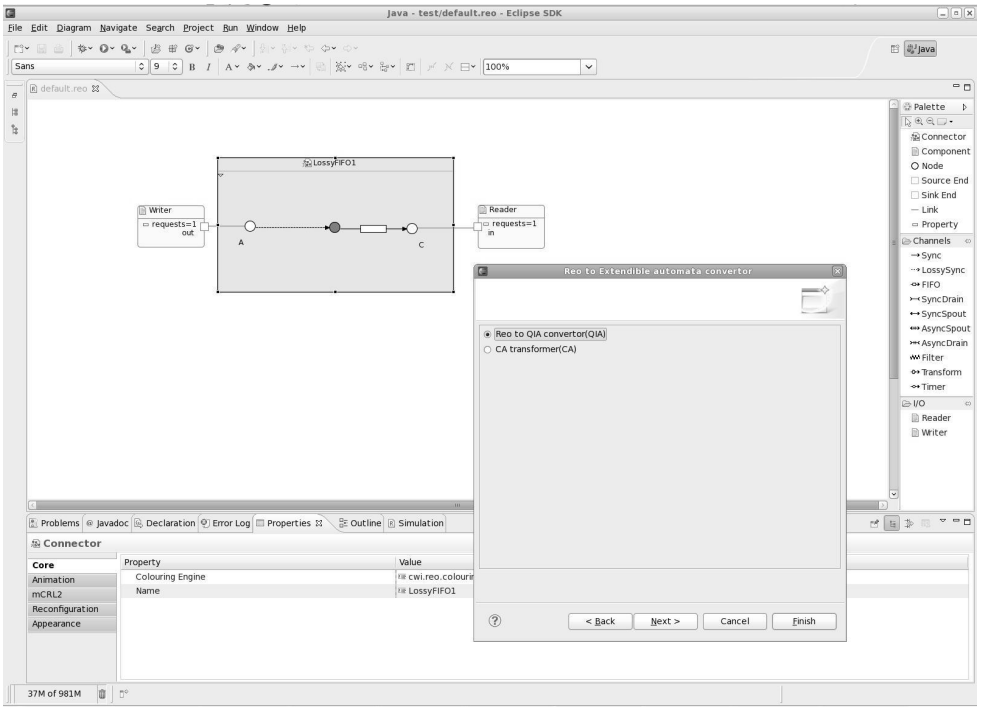


Figure 5.6: Translation of a Reo circuit into its QIA

corresponds to which node. Thus, if some nodes are not important (like mixed nodes), then the user does not need to give them names. For example, we can skip naming the mixed node of a LossyFIFO1 circuit, shown in gray in Figure 5.4.

Generating QIA for Stochastic Reo

CA and QIA are operational semantic models for Reo and Stochastic Reo, respectively. Reo2MC is a part of the ECT toolset, and the ECT supports converting a Reo circuit to a CA and a Stochastic Reo circuit to a QIA. As mentioned above, Stochastic Reo is an extension of Reo with the annotation of rates. In addition to that, setting rates can be postponed by setting all the rates with the default value 0.0. That is, any Reo circuit can be considered as a Stochastic Reo circuit. Users decide which semantic model will be the target of the translation. For converting a Reo circuit to its semantic model, "*Convert to Extensible Automaton*" must be chosen in the pop-up menu (which can be invoked using the mouse right-click on the circuit) of the Reo circuit; then the user can decide the file name for the conversion result and which semantic model to use as the target model for the conversion. Figure 5.6 shows the step of selecting the target semantic model.

Translation from QIA to CTMC

From a generated QIA model, we can obtain its corresponding CTMC model. For this translation, "*Translate to MC diagram*" must be chosen in the pop-up menu of a QIA model. The result of this translation is a state diagram of a CTMC model. Figure 5.7 shows the CTMC model derived from the QIA for the LossyFIFO1 circuit above.

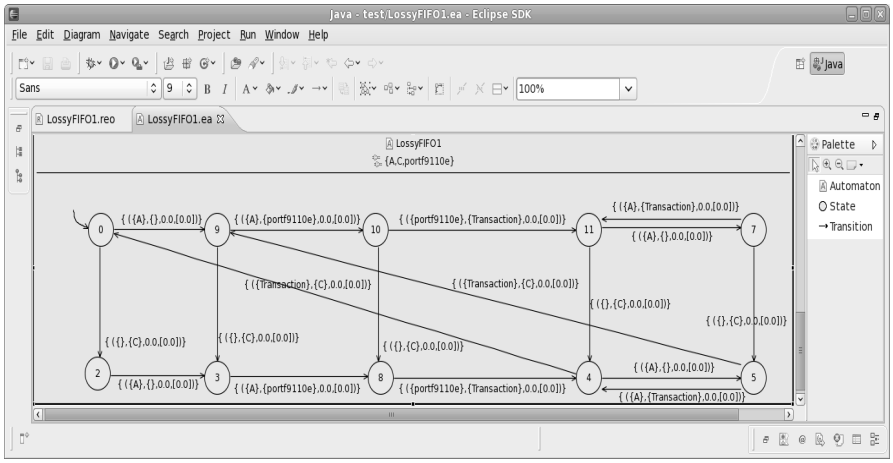


Figure 5.7: Generated CTMC model for LossyFIFO1 in Figure 5.4

For stochastic analysis, this result is fed into other analysis tools such as PRISM,

MATLAB, and Maple. For this purpose, Reo2MC supports generating the textual file describing the derived CTMC. For this generation, "*Generate MC textual file*" must be chosen in the pop-up menu of the derived CTMC model. In the window (see Figure 5.8), a result file name and its file format are specified in, respectively, *File name* and *Output file format* entries. Reo2MC supports the "*sm*" file format for PRISM and the "*csv*" file format for MATLAB and Maple.

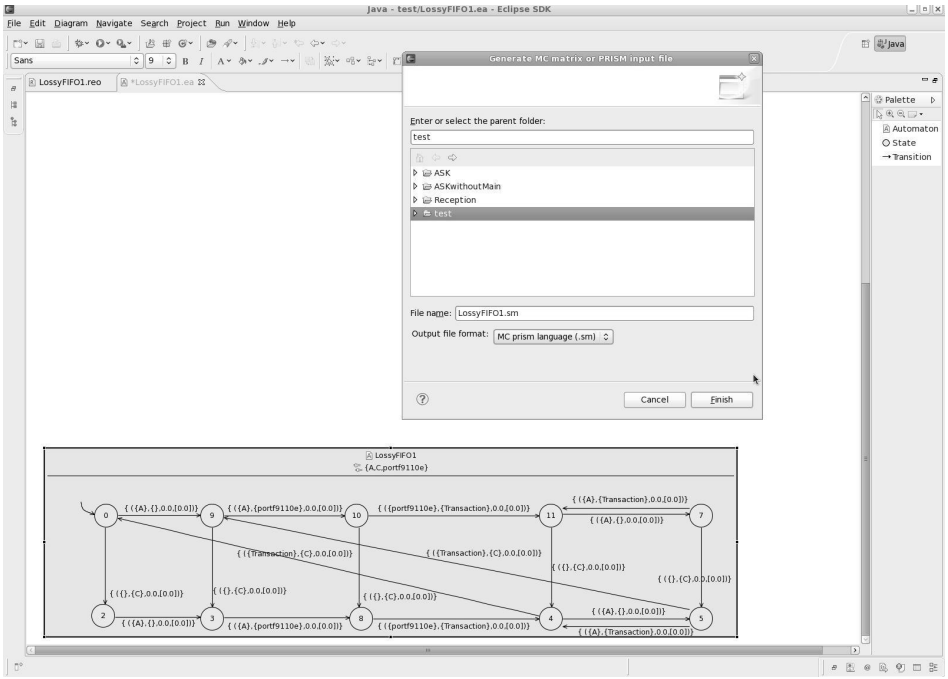


Figure 5.8: Generation of a textual file for the derived CTMC

Translation from Reo to CTMC

In general, the state space of MC grows drastically fast which causes the state-explosion problem. As shown in Figure 5.7, the result of the translation into CTMC models via QIA describes the whole state diagram. In the case of large systems, it may not be feasible to generate the state diagram of the CTMC for the whole Stochastic Reo circuit since, in general, it takes long to draw and deploy the whole state diagram. Moreover, the large graphical result of the translation is neither tractable nor readable. Thus, Reo2MC also provides the translation from Stochastic Reo circuits into the textual representation of the derived CTMCs without showing the whole CTMC diagrams. To skip drawing the state diagram of the derived CTMC, "*Convert Reo to Markov Chain - no diagram*" must be chosen in the pop-up menu of a Reo circuit.

Figure 5.9 shows the translation from the LossyFIFO1 circuit into its textual CTMC model for PRISM with the "sm" file extension.

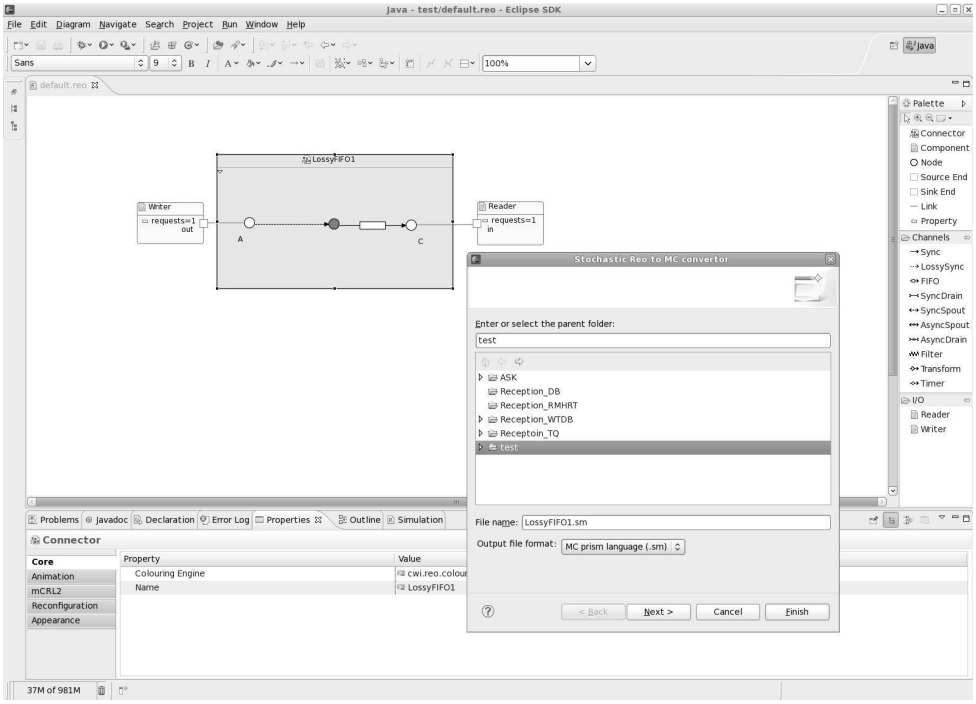


Figure 5.9: Translation of LossyFIFO1 into a CTMC model

Link to existing stochastic analysis tools The files generated by Reo2MC can then be fed to other tools. Figure 5.10 shows a file opened in PRISM, generated from the LossyFIFO1 circuit in Figure 5.9. As an example of stochastic analysis in PRISM, Figure 5.11 shows the graph of the variation of the probability of data-loss at node A in the LossyFIFO1 circuit when other rates of data-flow through the connector and I/O request arrivals at the other boundary node C are all set as 1. The graph shows as the frequency of I/O request arrivals at node A increases, the data-loss increases since node A is blocked more frequently.

5.3 Discussion

In this chapter, we introduced the Reo2MC tool in the ECT which is an integrated toolset for the design and the verification of Stochastic Reo. As a plug-in for the ECT, it provides the following functionalities: 1) editor for Stochastic Reo 2) generating

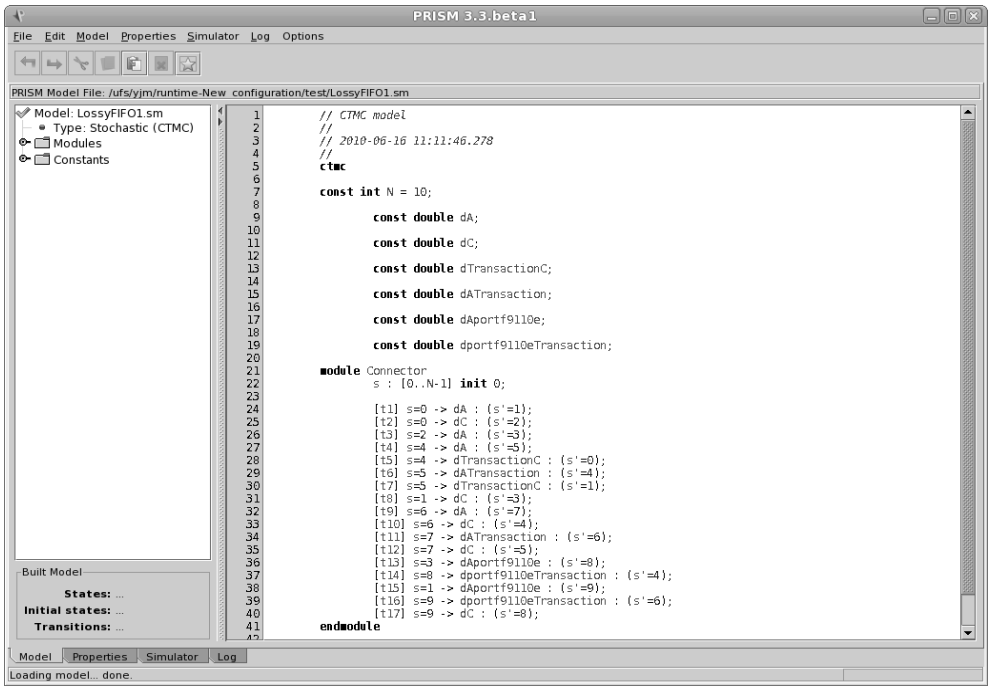


Figure 5.10: Using the PRISM with the generated file for LossyFIFO1

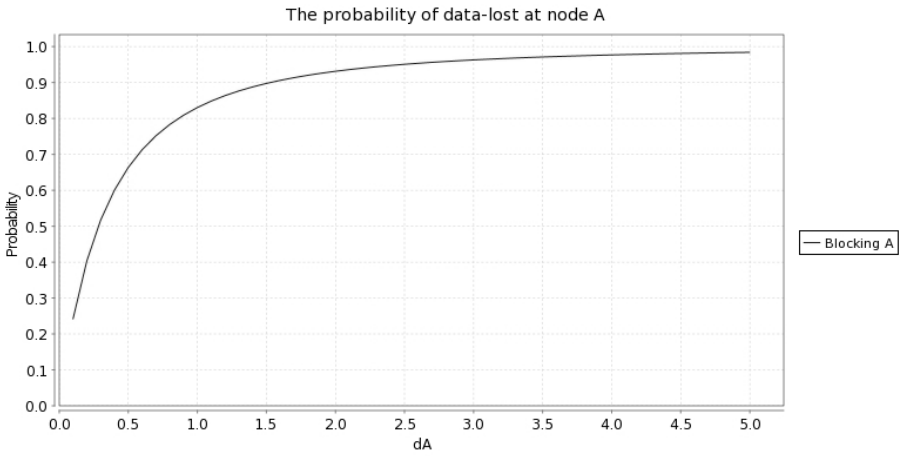


Figure 5.11: Example of using the PRISM

semantic models for Stochastic Reo, in particular, QIA, 3) deriving a CTMC model for a Stochastic Reo circuit.

Moreover, in this chapter, we also explained some implementation details of the Reo2MC tool. We showed the data structures of certain elements such as stochastic rates and delay-sequences: stochastic rates are used in Stochastic Reo and QIA; delay-sequences are used as an intermediate object for the translation from Stochastic Reo to a CTMC model.

Stochastic rates are annotated as properties of stochastic processes on a Reo connector modeling request-arrivals at channel ends and data-flows between channels. These constitute a delay-sequence for the translation from Stochastic Reo to its corresponding CTMC. A delay-sequence conveys the local topology of a connector, which is used to delineate synchronized data-flows. Thus, the data structure of Stochastic Reo contains the information about the topology of a connector.

The implementation is based on the definitions and the algorithms that we explained in Chapters 3 and 4.