An UML Profile for the specification of distributed component systems

Intern:	Advisor Professor	Co-advisor
Emil Salageanu	Eric Madelaine	Ludovic Apvrile

INRIA, OASIS Team,
Sophia-Antipolis,
September 2006

Table of Contents

1. Abstract

The objective of the **Oasis Project**, a common project between INRIA, I3S and UNSA, is to propose principles, techniques and tools for the construction, the analysis, verification and maintenance of systems in a distributed application context.

ProActive library [9] is a GRID middleware, developed by the OASIS team, for parallel, distributed and concurrent computing, also featuring mobility and security in a uniform framework. ProActive features a component-based programming using Fractal.

The Fractal model contains an Architecture Description Language (ADL), allowing the description of the structure of applications built from generic and reusable components. We have also proposed extensions to Fractal-ADL for attaching behavior specification in the descriptions. Lotos, a language for describing parallel communicating systems and FC2, a language for automata description, are thus proposed

But these languages are too low-level to be exposed to a non-expert developer. We intend to propose new languages, textual or graphical, that would be more abstract and easier to use.

Turtle Model, which we present in the next section, is an UML profile for modeling and formal validation of real-time systems. A tool for edition and validation of Turtle diagrams, TTool, was also implemented by the LabSoC laboratory from Telecom Paris.

Chapter 3 and 4 of this report contain a case study of a well-known example (Consumer-Producer). This example will be specified in two different ways: using Turtle profile and TTool editor in a first step, and using Component diagrams and State Machines Diagrams from UML2. We also propose a translation between the two models.

An extension of Tool, called CTTool was developed (an alpha version) for designing Component Based Systems. A description of this extension, a reference manual and also the design of our Producer-Consumer example within the new tool will be presented in chapter 5 of this report.

In chapter 6 we indicate what should be added or changed in this prototype (alpha version of CTTool) for creating a beta version which could be used for modeling real distributed-

component systems.

The final goal of our work is to extend CTTool (and eventually the Turtle model) to be able to manage and to create the formal specification for a ProActive system using high level constructions specifically adapted for distributed components.

2. Turtle and TTool

2.1 Turtle presentation

TURTLE is a UML profile dedicated to the modeling and formal validation of real-time systems [5].

One of the strength of the TURTLE profile is its formal semantics given by mapping from TURTLE diagrams to a specification in a temporal process algebra named RT-LOTOS or a LOTOS specification.

TURTLE defines the semantics for the following set of diagrams:

- An interaction overview diagram plus a set of sequence diagrams referenced by the interaction overview diagram. Such a set is called a "TURTLE Analysis" in TTool.
- A class diagram plus a set of activity diagrams: there must be exactly one activity diagram per Turtle Class defined in the class diagram. Such a set is called a "TURTLE Design" in TTool.

TTool can perform formal validation on each set defined just before.

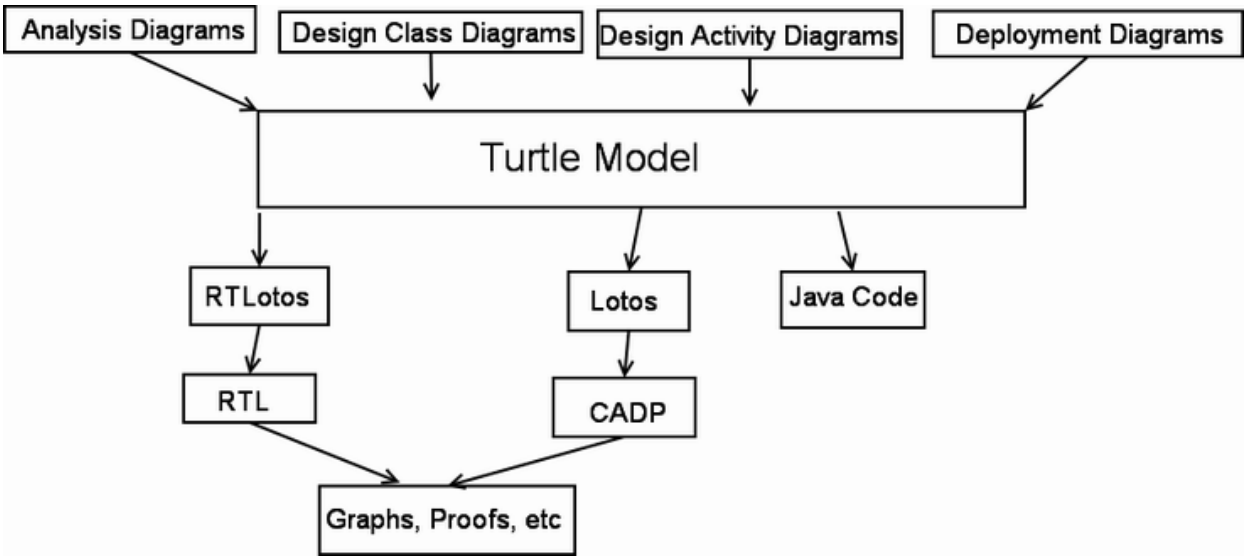


Figure 2.1 Turtle Model

The formal semantics of TURTLE makes it possible to automatically create (RT) Lotos code and perform formal validation without having any knowledge on the formal specification generated.

From the set of diagrams Turtle dispose we are interested in Class Diagrams and Activity Diagrams, those two being close to our modeling goals.

Also, amongst the formal specification formats Turtle is able to generate we are interested in the Lotos code. We can use turtle's facilities to generate the graphs or directly CADP tools.

We will illustrate these diagrams on our Consumer Producer example in the next chapter.

2.2 Adding new facilities to Turtle

At this moment Turtle doesn't have facilities to model component based systems. The goal of our work is to add two new diagrams to Turtle based on / close to Composite Structure Diagram and State Machine Diagram defined in UML2. We think that these two diagrams are the most appropriate for a ProActive model and easy to use by a Proactive developer.

In the section number 4 we will provide a detailed comparison between Turtle Diagrams and

UML2 based-on diagrams.

Composite Structure Diagrams are indispensable for a component-based system. Also, for the user/developer they are the most natural way to design the architecture of this kind of system.

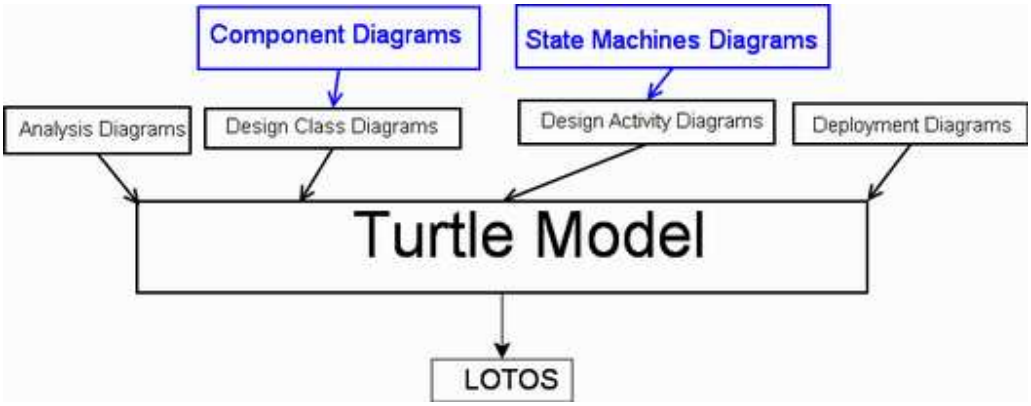


Figure 2.2 Adding new facilities to Turtle

We find State Machine Diagrams more appropriate to our model than Activity Diagrams. We can specify the comportment of our components hierarchic system with a State Machines and Submachine hierarchy. We can also specify methods call which, in Activity Diagrams need parallel operators which would induce an expansion of our model's complexity. We also find these diagrams more naturals for the final user.

In the long term, we want Turtle to automatically generate the diagrams for the components that are normally generated by ProActive (Queues, Proxies, etc.). The user would just put an ActiveObject symbol on his diagram and the tool will generate all diagrams specifying this object's Queue, Proxy, Default Body etc. The developer will only design the corresponding diagrams to his future code.

After defining all the operators we need for the two new diagrams we will have to translate them into the Internal Turtle Format. To do that, we try to define two semantic injections: from Component Diagrams to Class Diagrams and from State Machines Diagrams to Activity Diagrams.

We will first start our work by studying an example, and by modeling it in two different ways:

- using current Turtle Facilities (precisely, in our case, Class Diagrams and Activity Diagrams)
- using Component Diagrams and State Machines Diagrams.

These two approaches will help us to identify the operators we need and define the new diagrams which will constitute the basis for an alpha version of Tool+ , an extension of TTool (see chapter 5 of this report).

3. Consumer - Producer example

3.1 Example description

We consider the Consumer Producer System described in [1].

The system is composed of a single bounded buffer (with a maxSize capacity) a fixed number of producers and consumers. In the first step we design the system with one producer and two consumers. The producer feeds the buffer with one element at once and each consumer requests a single element per iteration.

For the beginning we consider that the buffer let a consumer waiting if it is empty and do the same with a producer if it is full.

We make the remark that the system is not parameterized in the number of producers and consumers but is parameterized by the size of the buffer.

We could easily also add parameters in the communication messages.

At the most abstract level, our system looks like in figure 3.1

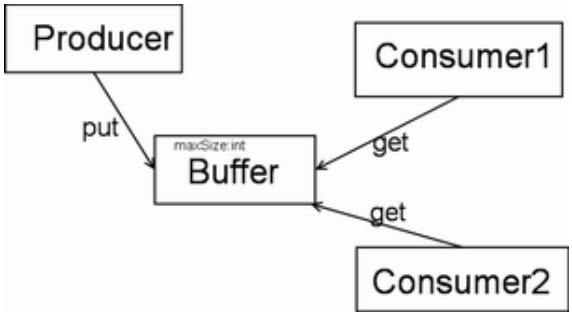


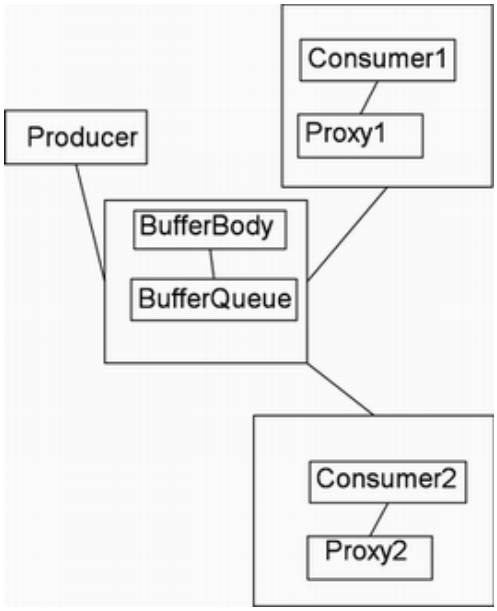
Figure 3.1 Consumer-Producer Example, abstract

3.2 ProActive approach

We will refine our model to make it closer to a ProActive model ([8], chapter 2).
We model the Buffer as two main components: a BufferQueue and a BufferBody.
We also add a ConsumerProxy to the consumer for managing remote asynchronous requests.

The producer sends a put signal (calls a put method) to the buffer. The buffer will put this request in his queue. The producer doesn't wait for any answer, he can continue its execution from the moment its put request is in the queue. The producer's proxy is trivial and can be omitted.

The Consumer sends to its proxy a 'get' request after which it continues its normal execution. The proxy will send this request to the buffer who will transmit it to the queue. Proxy is blocked, it waits for the answer. The BufferBody will treat its request at some moment and send the response (the element) to the proxy. At this time, the proxy allows the Consumer to use the value.



Remark. We didn't specify the creation of the 'future' in our model, not being concerned by the data transmitted from an object to another, but by the message sequences. Thus, we are interested in the fact that the consumer will need to synchronize with the proxy when he will use the value requested, and not in what this value might be.

3.3 Informal behavior description

We will describe the behavior of each of our primitive entities.

Producer :

- puts an element in the buffer. Repeat this operation indefinitely.

Consumer:

- sends a 'get' request to the proxy
- blocks only when it needs to use the requested element

ConsumerProxy:

- receives a 'get' request from the consumer.
- sends a 'qGet' request to the buffer
- waits for the 'rGet' response

- after receiving it allows the consumer to use the value.

BufferQueue: receives two kinds of signals:

get or put requests – it puts these requests in the waiting queue.

serveFirst, ServeFirstGet, ServeFirstPut – it takes the corresponding request from the queue and give it to the body for treat. The request is removed from the queue.

BufferBody: if the stock is empty it only treats put requests (oldest first).

if the stock is full it treats only get requests (oldest first).

else it treats the requests from the queue in FIFO order.

4. Modeling the Consumer-Producer example.

As we intent to design a new tool for modeling component distributed systems we will first model our example in the existing Turtle Design Diagrams in order to identify features that are not well adapted for the systems we need to model. Further in this chapter we will show a model of the same system using hand-made Composite Structure Diagram and State Machine Diagrams. Finally, in the next chapter we will present the new extension of TTool we have designed and implemented and also will present a complete modeling of the Consumer-Producer System designed with the new CTTool.

4.1 Turtle/TTool modeling.

4.1.1 Class Diagram.

The purpose of TURTLE class diagrams is to describe the interfaces of classes, and the relation between them. Interfaces of classes include regular attributes (boolean and natural types), and gates, which are the only way for classes to communicate with each others.

Classes may be connected together with association links.

A class diagram model models a set of tclasses that represent the structuration of the system user design. The behaviour of the system itself is described by means of composition operators and activity diagrams.

The TURTLE profile provides a formal semantics to relations between tclasses.

As we have already mentioned we can not design, at this time, component diagrams in Turtle, therefore in a first step, we have created the class diagram for our system.

For each class we have defined the attributes, the gates and have specified an activity diagram.

Defining gates. Whenever two instances need to communicate they can do that through specific gates.

When a gate wants to communicate (establish a message transfer) with another gate synchronized to it, it must wait until the other gate is ready to synchronize.

Also, if two instances of an object B send the same signal to an object A, A must have two gates for this signal, one for each instance of B. We can also call them ports.

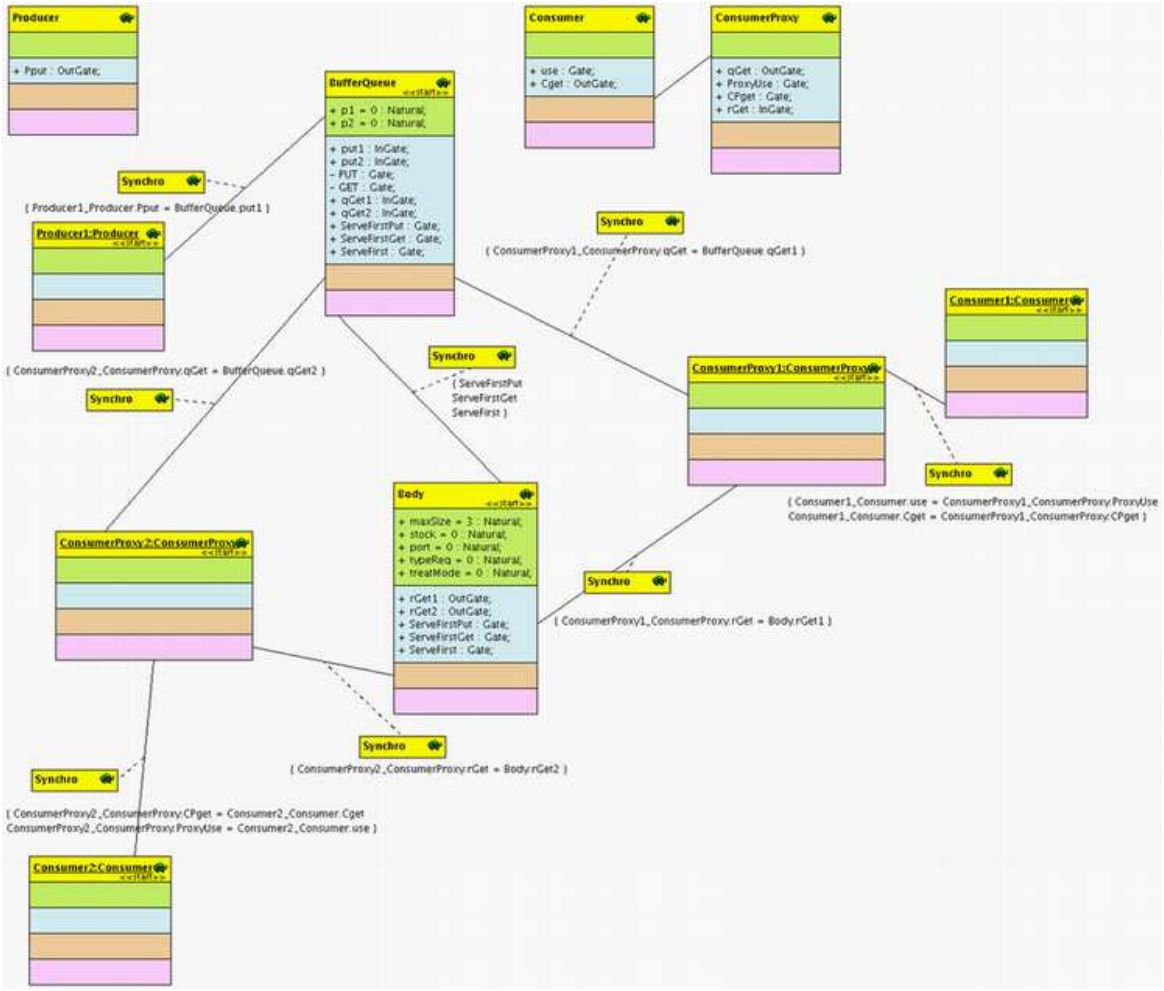


Figure 4.1 Turtle class diagram

In Turtle a method call is defined as a synchronization of two gates.

In our example:

- The producer synchronizes with the BufferQueue on the pair of gates : Producer.pPut – BufferQueue.put1.
- The first instance of ConsumerProxy synchronize with the BufferQueue on the pair of gates ConsumerProxy.qGet – BufferQueue.qGet1.
- The second instance of ConsumerProxy synchronize with the BufferQueue on the pair of gates ConsumerProxy.qGet – BufferQueue.qGet2.
- BufferQueue will know who sent the request because it receives it on different gates.
- The proxy will synchronize with the BufferBody to get the response.

For a better understanding of the model specification the reader of this report can load into TTool (or CTTool) the model of the Consumer-Producer system witch can be found in the 'modeling' folder of the latest released of CTTool.

4.1.2 Activity Diagrams.

The purpose of TURTLE Activity diagrams is to fully describe behaviors of tclasses.

Thus, TURTLE activity diagrams offer three kinds of elements:

- Connectors.
- Logical operators. This includes actions on gate, actions on attributes, choice, and four other operators.
- Temporal operators.

In order to model our system we have designed 5 objects, with one class and one activity diagram for each of them.

For example, the body activity diagram models the comportment of the body object. We have initialized size attribute with 0 and maxSize attribute with 3. The activity diagram states that, if the size is 0 (we don't have any element in the buffer) we will only treat 'put' requests from the queue; if size is maxSize we will treat only 'get' requests, otherwise we will treat FIFO.

In order to treat a request the body synchronizes with the BufferQueue on specific gates, depending on what strategy it uses to treat requests.

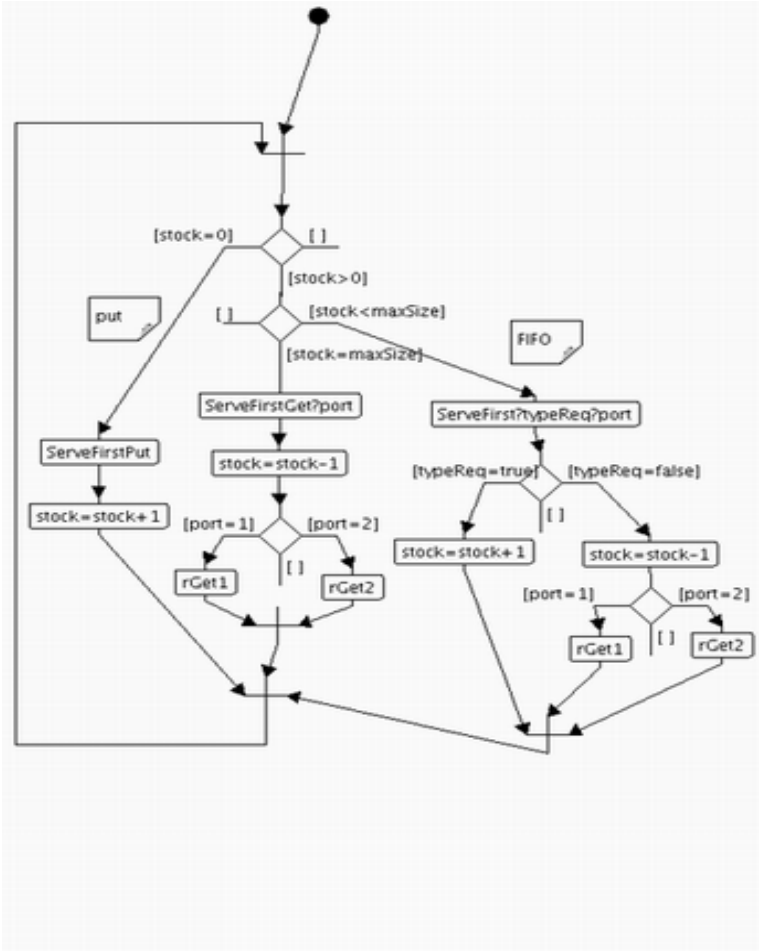


Figure 4.1.2 Body Activity Diagram

When it treats 'get' requests it receives from the queue the port corresponding to the consumer which sent the request and to whom the answer will be sent.

4.1.3 Turtle diagrams evaluation against a Component Based System

Class Diagrams.

- The main entity in the Turtle Class Diagrams is the Class. Classes and instances cannot be hierarchically composed so a Class diagram cannot offer the possibility of modeling a hierarchical component system.

For example, in our previous model, we have modeled the Buffer Component as two classes - BufferQueue and Body instead of one component with two sub-components.

- A Class Diagram offers the possibility to define gates and synchronize them to establish communications. We need to model components with communication ports, define communication interfaces in order to bind components to each other. Thus, the messages exchanged between components will not be defined inside the components themselves but inside the interfaces.

- We also need to have a flexibility for components (e.g. subtyping) the Turtle Class Diagram do not offer.

Activity Diagrams

- Unlike Activity Diagrams, State Machines Diagrams offers the possibility of a modular behavior specification by defining sub-machines.

- Besides, in the Activity Diagrams, we cannot specify the port a message comes from or is meant to go to in order to identify the sender or recipient of this messages. For example, in 'Figure 4.1.2 - Body Activity Diagram' we used to different gates to manage the sending of the same message to two different entities. What we would like to do is specifying a port where the message is sent without being concerned on the recipient of the message. Activity Diagrams do not allow this specifications.

- We also want to replace gate synchronization with receiving and sending of messages which is more appropriate to our systems.

4.2 UML2 Composite Structure Diagrams and State Machines Diagrams

Before describing our alpha version of the component-based extension of TTool, CTTool, we give a UML 2 [3], [7] approach of our Consumer-Producer example. Diagrams in this section are hand-made, their goal is to show our requirements for a new tool.

We will now model our system using component diagrams and state machines diagrams. This approach is meant to be close to what the final user of the new Tool will have to design.

4.2.1 Composite Structure Diagrams

In [3], Chapter 8, a component is specified as a modular unit with well-defined interfaces that is replaceable within its environment.

A component can be considered as an autonomous unit within a system or subsystem. It has a set of provided and required interfaces exposed via ports and its internal can only be accessible through these interfaces. As a result, components and subsystems can be flexibly reused and replaced by connecting them together via their interfaces.
Therefore, a component type conformance is defined by its provided and required interfaces.

Two types of components are specified in [3]:

• **Basic Components** (we will refer them as **Primitives**). A component is specified as an executable element in a system. A component is a specialized class that has an external specification in the form of one or more provided and required interfaces. One or more classifiers realize a component's behavior.

• **Packaging Components** (we will refer them as composites). A component is specified as coherent group of elements as part of the developing process. It can own and import a set of model elements.

In our diagrams we did not defined realizing classifiers, which means that a component on the diagram will also be its unique instance and its behavior will be specified in its State Machine Diagram.
In figure 4.2 we have model the Consumer-Producer system as four main entities: a Producer, two consumers and a buffer. For each component we have specified in and out ports, required and provided interfaces. This is only a case study, a complete model of the system (made within CTTool we have implemented) is described in Chapter 5 of this report.

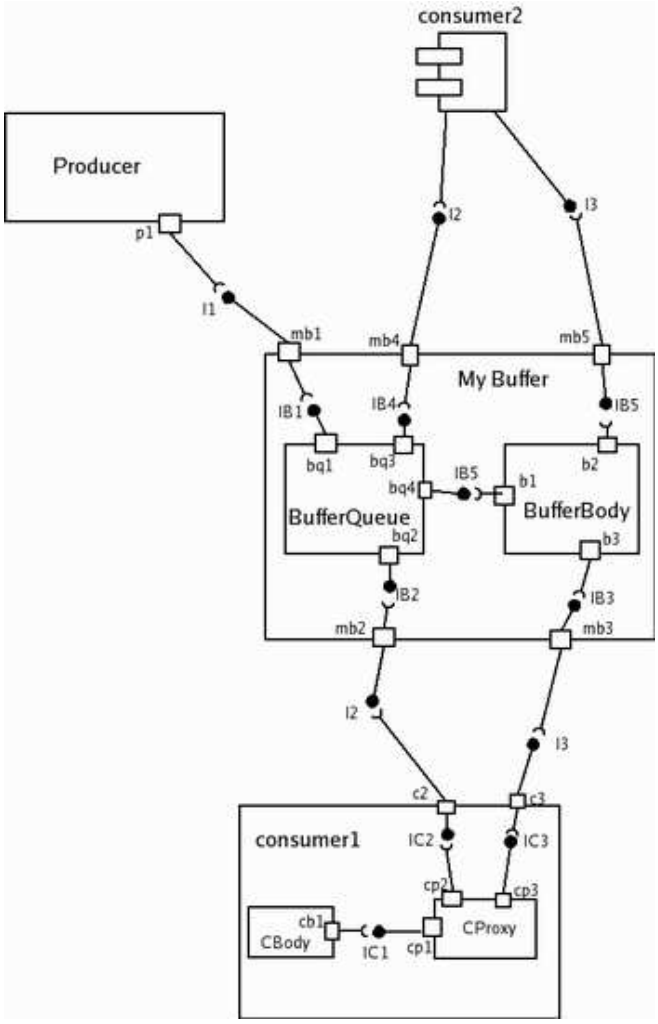


Figure 4.2 Component Diagram

4.2.2 State Machines Diagrams

The State Machines Package, [3] Chapter 15, defines a set of concepts that can be used for modeling discrete behavior through finite state-transition systems. Two kinds of state machines are defined: Behavioral State Machines and Protocol State Machines. We are interested in the first ones.

State machines can be used to specify the behavior of various model elements.

For our Consumer-Producer Example we specify a state machine corresponding to each primitive component from the component diagram.

Also, for each signal we will specify the port this signal is sent to or received from.

We exemplify with the state machine for BufferBody primitive component.

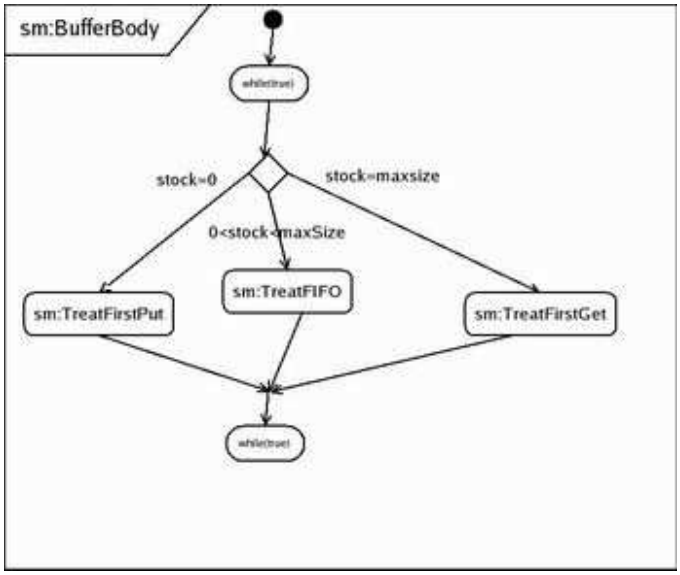


Figure 4.2.2 BufferBody State Machine Diagrams

As we see, the BufferBody choose on of the treatment strategies depending on the actual stock value.

4.3 From UML2 Component Diagrams and State Machine Diagrams back to Turtle

In [6] few ideas about how to make Turtle evolve are given. We present in this section few guidelines in making a transition from a component-based model to Turtle model. An alpha design and implementation of a Turtle component-based extension, CTTool is presented in the next section, as well as a technical report.

As we have already mentioned in the beginning of this report, we need to define a translation, at a semantically level, from Component Diagrams and State Machine Diagrams to Turtle Class diagrams and Activity diagrams.

In this way we will be able to use Turtle facilities to generate graphs and make proofs for our model.

Some observations need to be done before.

Our translation is made with loss of information, especially concerning the component structure which will be translated by a simple class diagram. We will also loose all information on the interfaces and ports that will be translated in Turtle gates.

As first consequence of previously remark, in this state of our work, we cannot manage non-functional interfaces (bind, unbind, etc.).

Also, we can not manage multiplicity – we need to define statically the exact number of instances of each class.

4.3.1 From Component Diagrams to Turtle Classes

Remark: The pseudo-algorithms proposed in this section are not exhaustive but just a case study of our Consumer-Producer Example. Further work needs to be done.

1. *For each primitive component we create a Turtle empty class. If the component has multiple instances we create the corresponding Turtle TObjects.*

2. *We merge similar interfaces and remove "forwarding" ports.*

Example: (See figure 4.3.1)

In our Consumer-Producer component model the composite MyBuffer will be removed

when translating to Turtle classes. The subcomponent BufferQueue will 'get out' from the composite MyBuffer. We call mb1 a 'forwarding port', it forwards the 'put' request from producer to the BufferQueue. Also, the interfaces I1 and IB1 are similar. We remove the port mb1, and merge the two interfaces like in figure 4.3.1

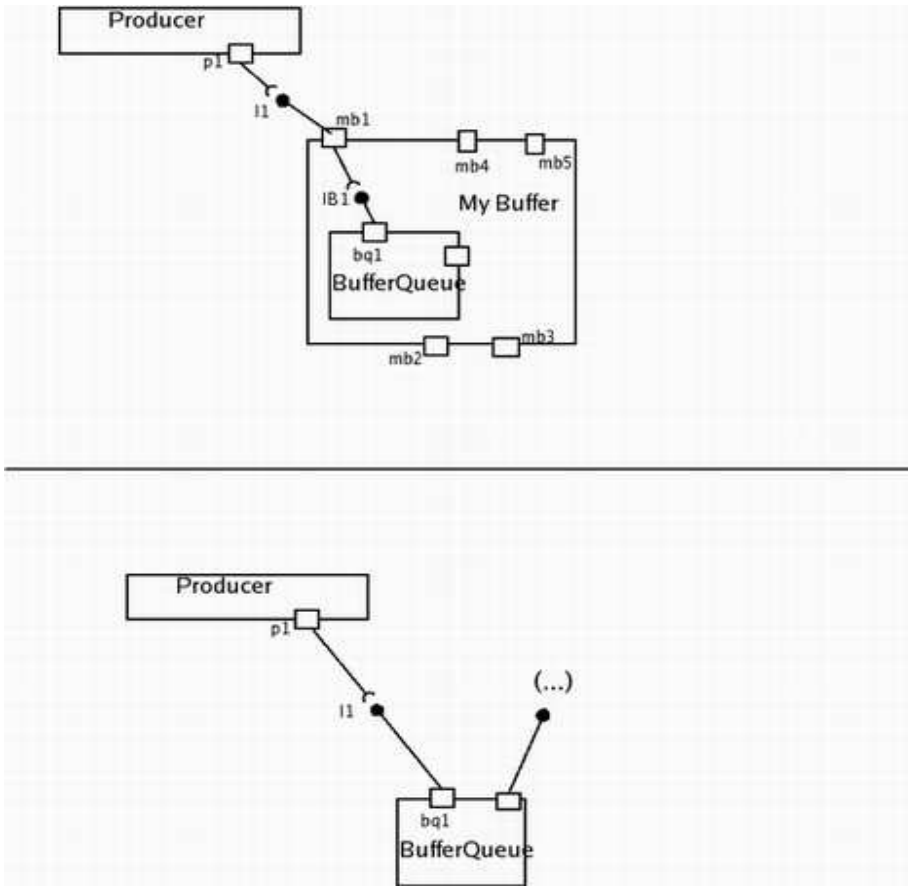


Figure 4.3.1 From component to classes, merging interfaces, removing ports

After having done all the operations similar with the previously, our component diagram becomes a diagram composed by only primitive components, like in Figure 4.3.2

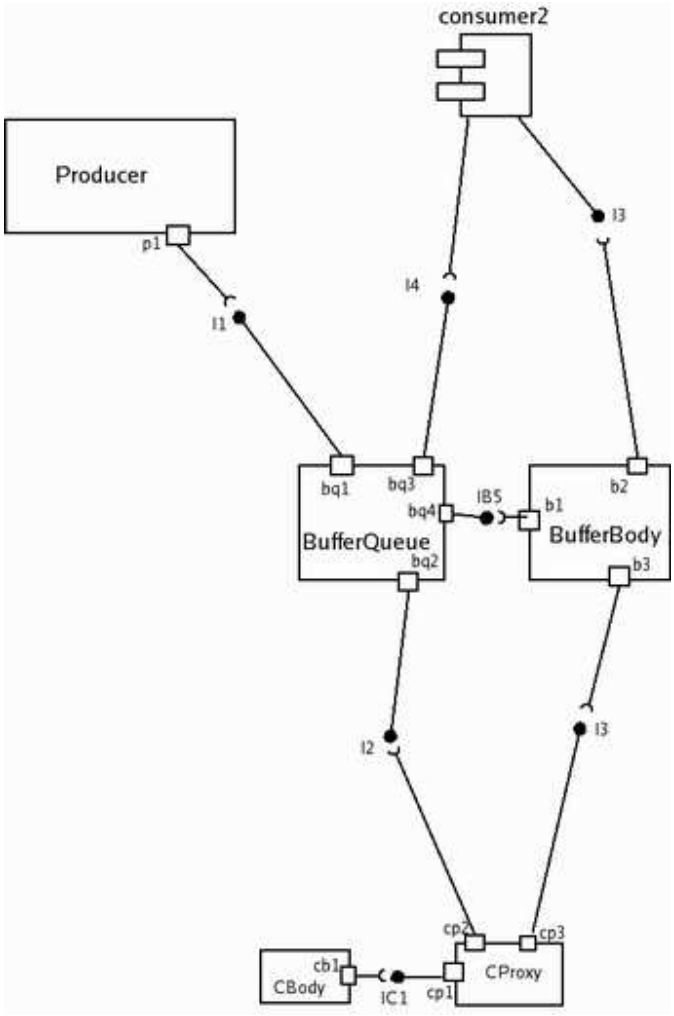


Figure 4.3.2 Primitive components

3. *Now we add to each port the methods of the interfaces offered or expected on it.*

4. *Then, for each component, for each pair (port, signal/method) we create a corresponding gate for the turtle class and define the corresponding synchronization.*

Example:

Through the port p1 of Producer pass the 'put' signal who gets in the bq1 port of BufferQueue.

We create an 'Out Gate' in Turtle's Class Producer (let's call it p1.put) and an 'In Gate' (bq1.put) in the class BufferQueue. We synchronize the two classes on the new created gates, or, if the classes are already synchronized, we add the two gates at the synchronization gates

set for the two classes.

So, we have created the Turtle Classes, the public gates and the synchronization between them.

4.3.2 From State Machines Diagrams to Turtle Activity Diagrams

Remark: Operators used in the Machines Diagrams must be formally defined. We must also choose which operators we will allow our user to use, and if we need extra operators. As we have mentioned, this paper is nothing but a study case.

1. *We will analyze the State Machines Diagrams to complete Turtle Classes and to create Activity Diagrams. In a first step, we will replace the submachine instances by actually copying the entire submachine inside the machine that uses it. The semantic definition of the Submachine allows us to do this operation.*

2. *Then we will map the state machine operators on action diagram operators:*

signal receiving -> action state

signal sending -> action state

junction -> junction

condition - > deterministic choice

machine state -> labeled junction

connector to a state -> connector to the corresponding junction

split/waiting different signals -> nondeterministic choice

...(not exhaustive)

5. CTTool, an extension of TTool

5.1 CTTool presentation

5.1.1 Introduction

We have developed, as a result of our research work presented in this report, an extension of TTool, called CTTool, which allows the user to model Component Based Systems.

We will present in this section the alpha version of CTTool.

CTTool defines two kinds of diagrams, Composite Structure Diagrams for architecture description and State Machines Diagrams for behavior description. The tool is able to translate the user model into Turtle Model from which the formal code will be generated.

5.1.2 Composite Structure Diagrams

Composite Structure Diagrams we have implemented in CTTool are based on diagram specifications in [3] Chapter 8 that we have introduced in Chapter 4.2.1 of this report.

Composite Structure Diagrams within CTTool. Diagram elements overview.

- **Component.**

We can distinguish two kinds of components: primitives and composites. A composite can have several internal components that can be primitives or composites.

Connections between components can be defined as well as interfaces in order to specify the communications that can occur between components.

A Component behavior must be described with a state machine diagram.

- **Port**

A port, [3] Chapter 9, is a point for conducting interactions between a component and its environment. Constraints may be defined for a port so that the owner component could be reusable within any environment that conforms to the interaction constraints imposed by its ports.

CTTool defines 3 kinds of ports:

- **inPort.** A provided interface can be exposed via an inPort. Through these kinds of port a component can receive messages from its environment. An answer can be sent back through the same port in the case of a synchronous call.

- **outPort.** A required interface can be exposed via an outPort. A component can send messages through an outPort and eventually receive an answer through the same port.

- **delegatePort.** There are two possible cases when a delegatePort can be used:

  - a message received in an inPort of a component C will be transmitted to a delegatePort of a subcomponent SC of C in order to let SC treat this message.
  - a subcomponent SC of a Component C needs to send a message in the enviroment of C. SC will send the message through a delegate port (of SC). The message will pass then in an outPort of C.

For a better understanding of ports behavior see 'message transit' in the example in this section.

- **Connector. Interface.**

We define to kinds of connectors between components:

- Connector from an out port to an in port - to connect two components (at the same level)

- Connector between an in/out port and a delegate port (to delegate messages to/from an inner component).

An interface is associated to a connection between two components. An interface defines a set of methods, each method can return or not a result.

Note. A connector is always oriented. Let C be a connector between a port p1 and a port p2 with the direction p1->p2. Then, one of the next possibilities must be true:

- p1 is an outPort and p2 an inPort. Only the owner of p1 (or one of its subcomponents) can initiate a communication.

- p1 is an inPort and p2 a delegatePort. A message can be received via p1 and sent forward to p2. Neither the owner of p1 nor the owner of p2 can initiate a communication.

- p1 is a delegate port and p2 is an outPort. The owner of p1 (or a subcomponent) can initiate a communication.

Version alpha does not implement the component type (defined by offered and required interfaces). In this version the user only specify the ports for a component and then, to each connection between two ports an interface can be associated. So the interface is not specified at the creation of the component but when a connection is created.

Remark: this way of connecting components (specifying an interface for a connection and not for the component itself) prevents one of defining a component-type as specified in 4.2.1 of this report and in [3] chapters 8 and 9. Allowing the developer to specify the interface at the connection time means that a component could change any time its type by simply changing the interfaces associated to its connections. Thus we cannot define the conditions under which a component can be replace within its context. Beta version should define provided and required interfaces for a component that will allow the user to bind components together and also replace a component within its context.

Synchronous/ Asynchronous message.
We call a message m "**synchronous**" if the sending of m and the reception of m occur in the same instant. In addiction, time line for the sender and receiver of m must be measured with the same time clock. In other words, we need to have defined a global time for the involved entities.

We call a message m "**asynchronous**" if the sending of m and the reception of m do not occur in the same instant or if there is no global time for the involved entities.

- **Attributes.**

A set of attributes can be defined (and, if needed, initialized) for each component. These attributes will be visible in the state machine diagram. For the moment, only Integer type was implement for attributes.

Example: Constructing a Component Diagram for the Consumer Produce Example.

First we define the three main components: Producer, Consumer and Buffer.

We add the ports and interfaces through which this components communicate to each other.

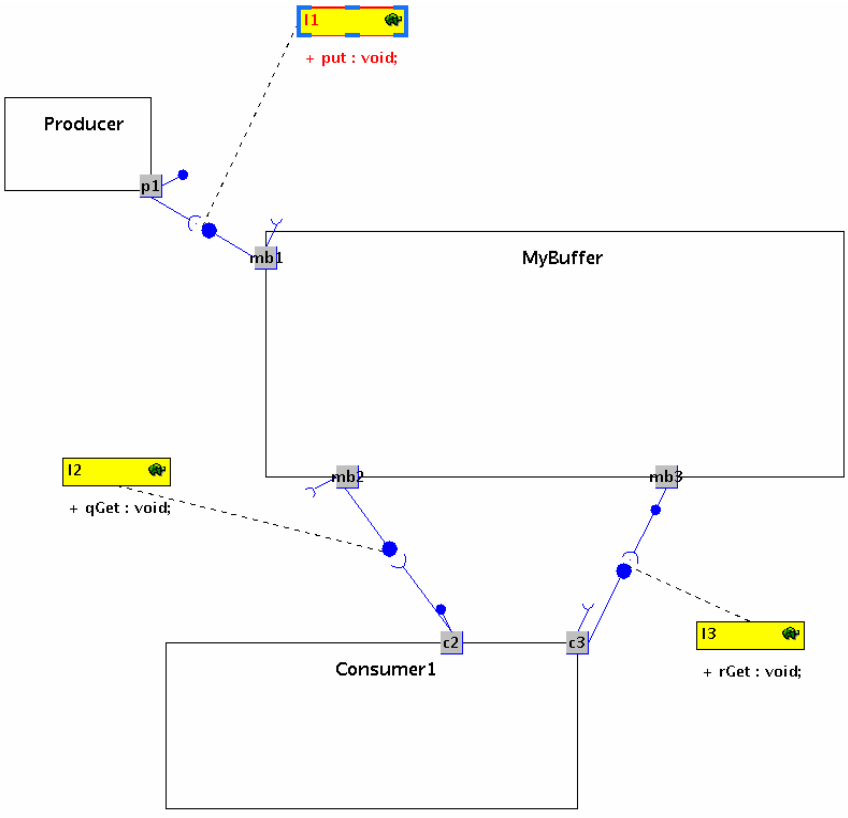


Figure 5.1 . Example: Constructing a Component Diagram for the Consumer Producer System.

In this diagram:

Producer

- can send "put" messages via port p1 (i.e. call a "put" method to MyBuffer.)

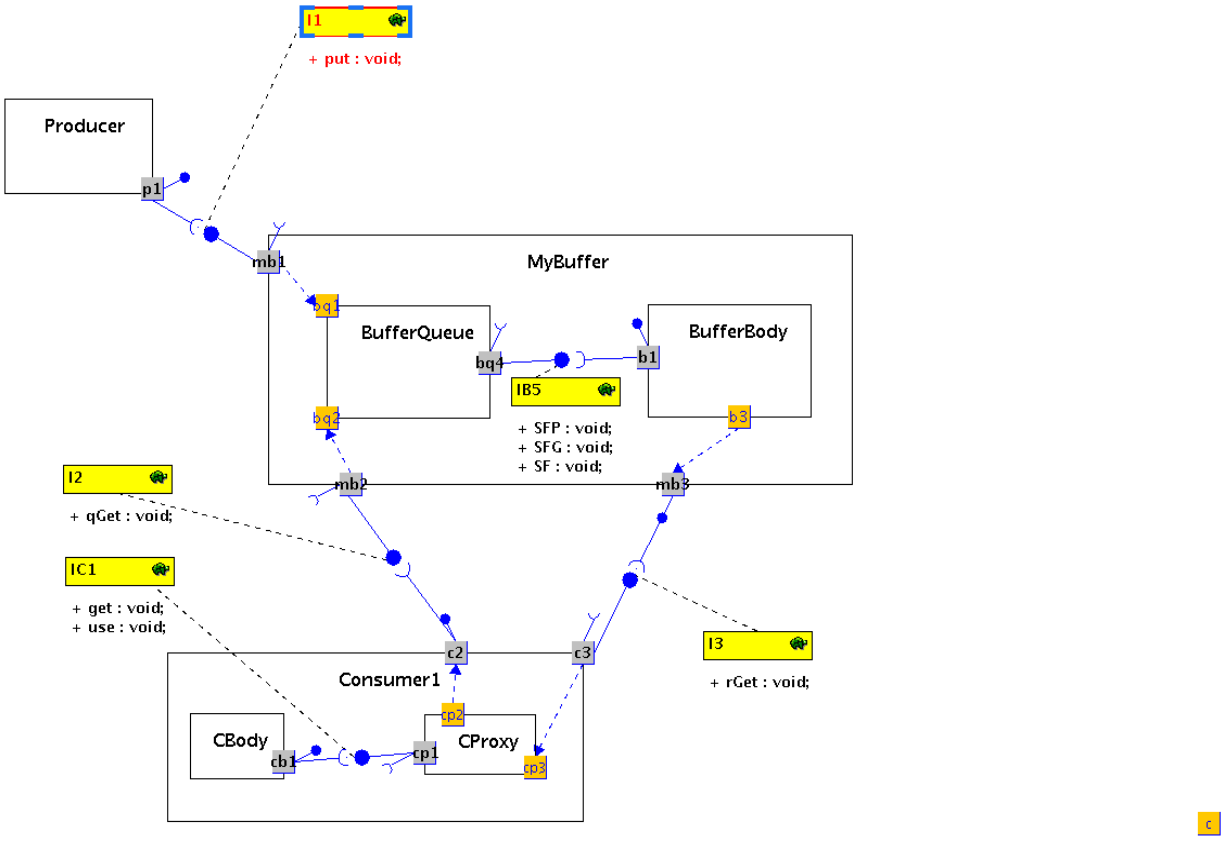
Consumer1

- can send qGet messages via port c2.
- receives rGet messages via port c3

MyBuffer

- receives "put" messages via mb1
- receives "qGet" messages via mb2
- sends rGet messages via mb3

We must now specify the "content" of MyBuffer and Consumer1 - the primitives from which this components are composed:



We defined the MyBuffer component as composed from BufferQueue and BufferBody primitives. The requests from the producer and consumer will wait into the queue to be treated by the body.

Consumer1 is composed by CBody and CProxy. The body sends a get message to the proxy and, in a second time it will try to use the requested information.

The messages transit on previous example

The message "put" sent by the Producer will arrive into port mb1 of MyBuffer. From this point it will be sent to port bq1 of BufferQueue through a delegate connection. Therefore the primitive BufferQueue will treat this message.

The message "get" sent by CBody via cb1 will be received by CProxy (via cp1). The proxy will send at this moment a "qGet" message through port cp2, message that will be sent forward by delegation to the father Consumer1 and will finaly reach port bq2 of BufferQueue.

Inside the MyBuffer Component the requests waiting in the Queue will be recuperated by

BufferBody who will treat them and send responses ("rGet" messages) via port b3. This message will transit to port cp3 of CProxy.

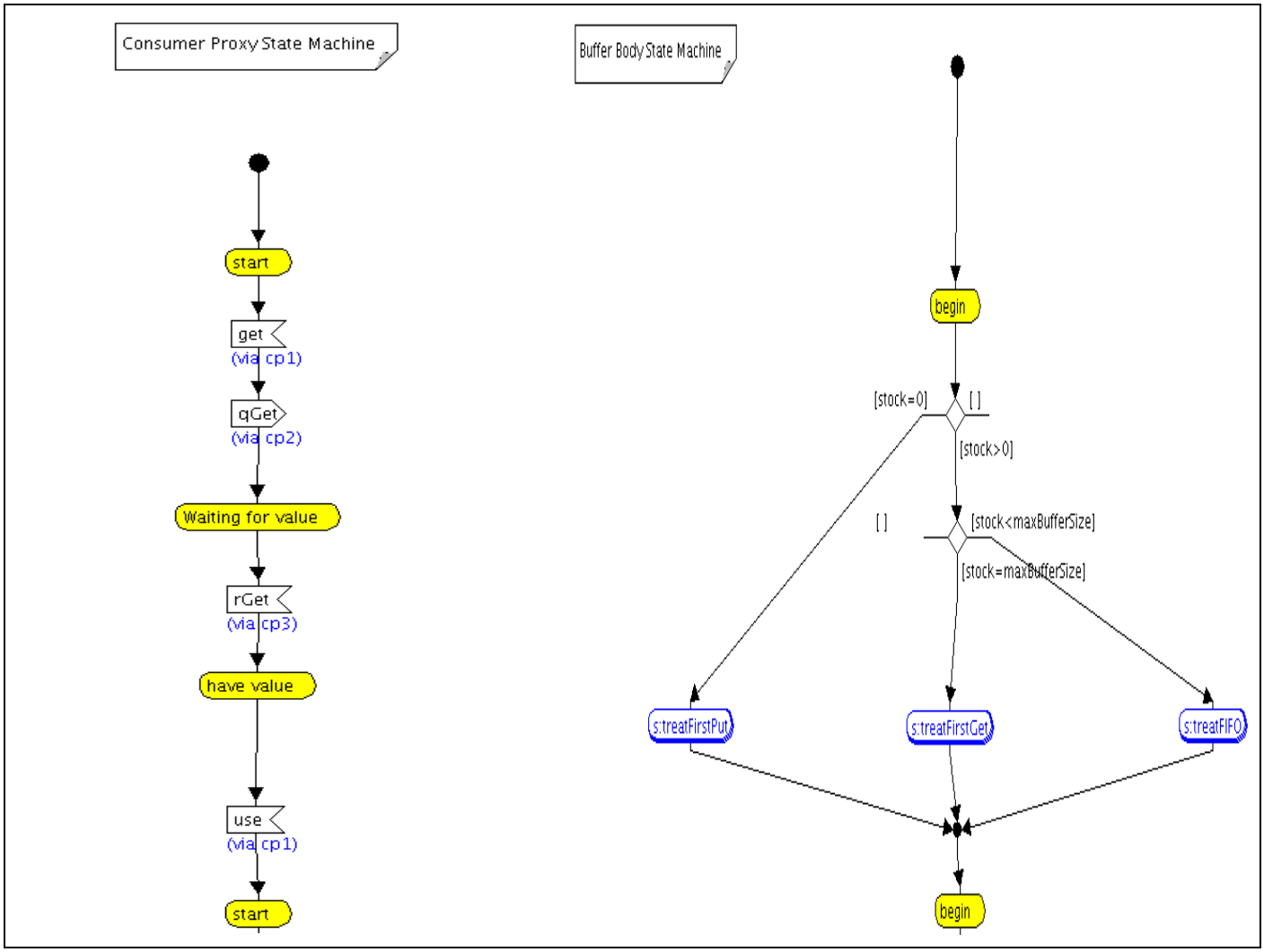
Remark: In this version alpha of CTTool we did not define asynchronous messages (beta version will offer them) but we can "hand design" asynchronism as sequence of synchronous calls.

For each primitive (Producer, BufferQueue, BufferBody, CBody,Cproxy) we must describe behavior by defining State Machines Diagrams.

5.1.3 State Machines Diagrams

Instead of Activity Diagrams offered by TTool, CTTool propose **State Machines Diagrams** that we find more appropriate for component design. They allow user to use "send message" and "receive message" operators, to specify the ports the messages are sent or received through, to define sub-machines.

We offer two examples of state machines.



State Machine Diagram operators.

"Send message" / "Receive Message" operators. A message and a port must be specified. The message must be defined in the interface associated with the port.

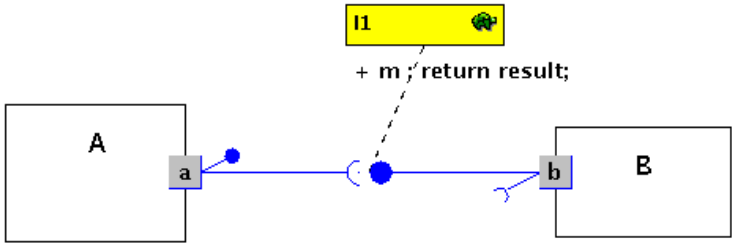
The message has the same format as the "action on a gate" in TTool:

<name of the message>(<!expr>*<?variable name>*)*

We asusme that expr is of any type. It can be a variable, a set of operations involving several variables, etc.

Examples:

Let m be a message defined by an interface associated to a connexion from port a of component A to port b of component b:



In State Machine of "A" :

- ***send message*** of form:

- ***m!x (via a)*** -> a message m is sent through port a from A to port b from B with a parameter x

  To this message should correspond a ***receive message*** in the State Machine of B:

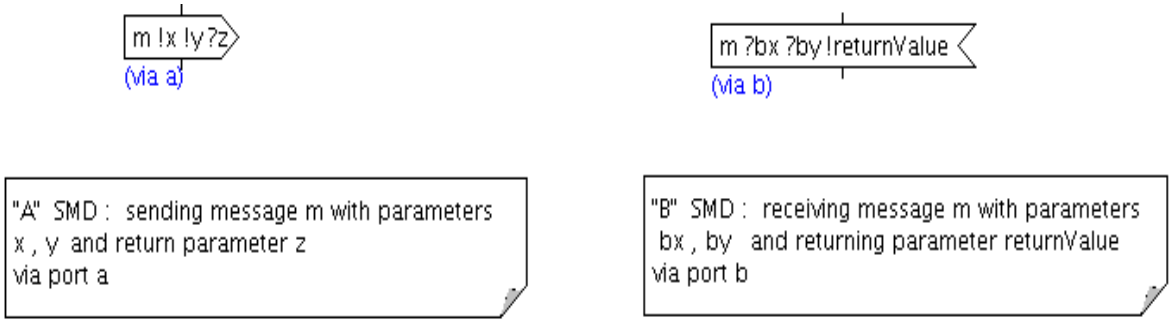
- ***m?t (via b)*** -> receiving of a message with a parameter

- ***send message*** of form:

- ***m!x!y?z (via a)*** -> sending m message through port a with parameters x,y and return parameter z

  To this message should correspond a ***receive message***:

- ***m?xb?yb!returnValue***



Semantically, it involves a synchronization: the send message and the receive message will have place at the same moment. SM of a will block until SM of b will be ready to receive the message.

Choice operator - has three possible branches guarded or not.

A branch not guarded is considered to be "true".

(to review the content of this phrase :)

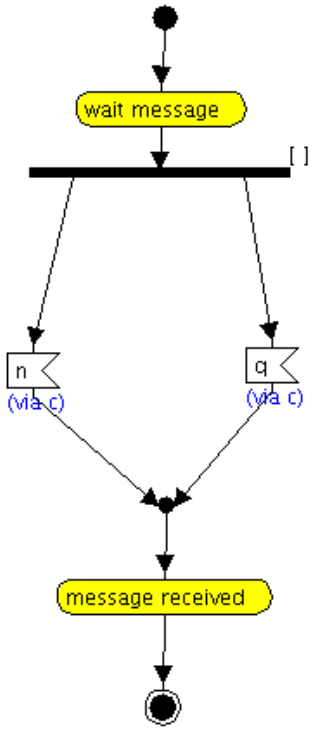
If more than one branch is true (more than one sub-activity may be executed) the choice between them is performed according to the first message received or which is successfully sent on one of the branches.

If more than one message can be sent or received at that point (which could be the case, we don't use time operators) we have a nondeterministic choice. We will see all the branches on the accessibility graph as possible transitions.

Nondeterministic choice operator.

Several messages could be received when a machine is in a certain state. We will use in this case a nondeterministic choice operator to which all "receive messages" will be connected.

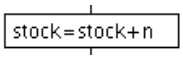
Only the first income message will synchronize (will be received) and will change the state of the machine.



In the state machine in the left two messages are waited.

Once one of the messages is received, let's say "n via c", the machine will change into the "message received" state and will not accept "q via c" message anymore.

Action Operator - specify an action involving local attributes (attributes defined in the component description on Component diagram).



Component Diagrams and **State Machine Diagrams** allow a complete description of a system and it's behavior. Once those diagrams defined one can make a syntax analysis which can detect some structure and behavior errors. Then Lotos code can be created and verified. Extern model checkers can be used within the tool to analyze the generated formal code (see TTool Online Help for mode details).

5.2 Consumer Produce System. CTTool Design - Complete Example Description

We consider the Consumer-Producer system presented in Chapter 3, page 7, of this report. In order to mimic a ProActive Model we will specify an asynchronous call from a Consumer and the Buffer.

The Consumer will send a qGet message to the buffer (which means it requests an information) and, later, it will receive a message rGet containing that information.

5.2.1 Composite Structure Diagrams with CTTool

We can start drawing our Component Diagram with TTool.+

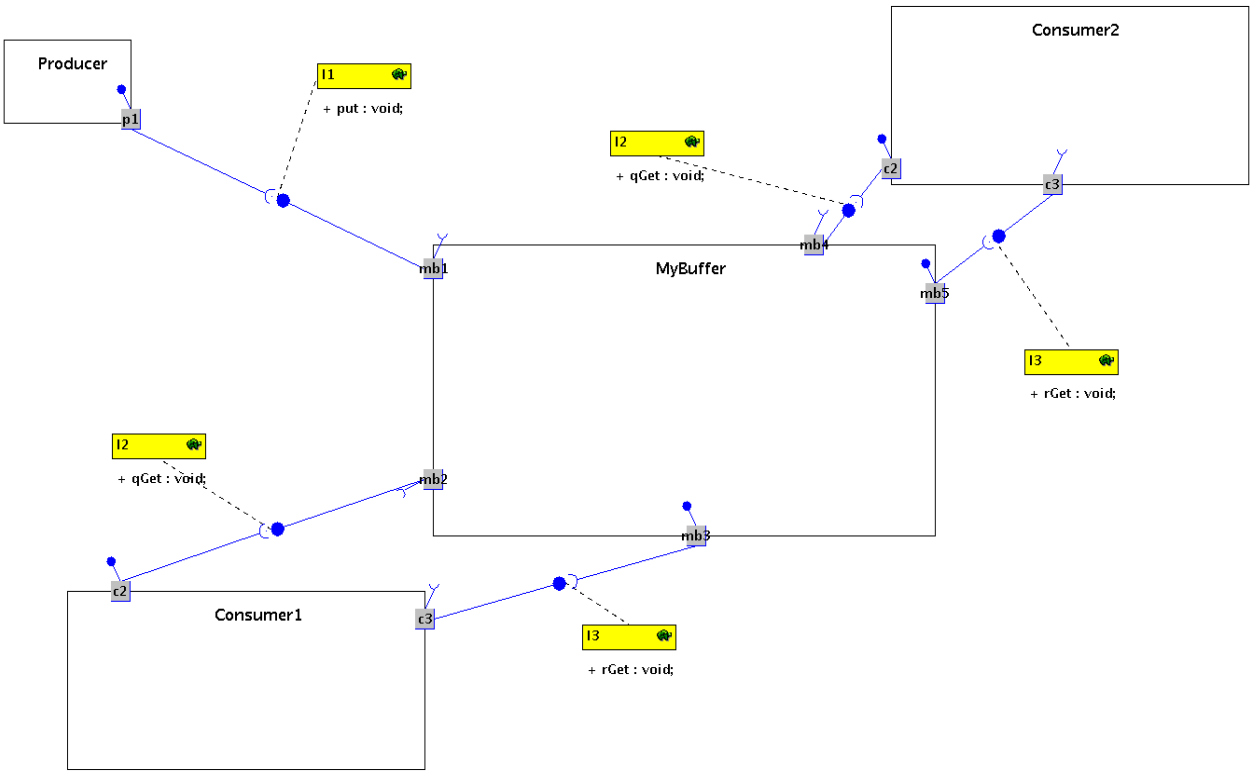


Figure 5.2.1. Designing main entities of Consumer-Producer system with CTTool

We have design the main entities involved in our system: one producer, two Consumers and a Buffer.

We have also specified the interactions between them by designing the communication ports, the connections and offered/required interfaces :

- the producer can send messages to the buffer through the port p1 (those messages will be received through the port mb1 of the buffer). This messages are defined in the interface I1. Actually it is only the method "put".
- each consumer can send to the buffer a "qGet" message and receive an "rGet" message.

We need now to design the "content" (internal components) of the components on the diagram:

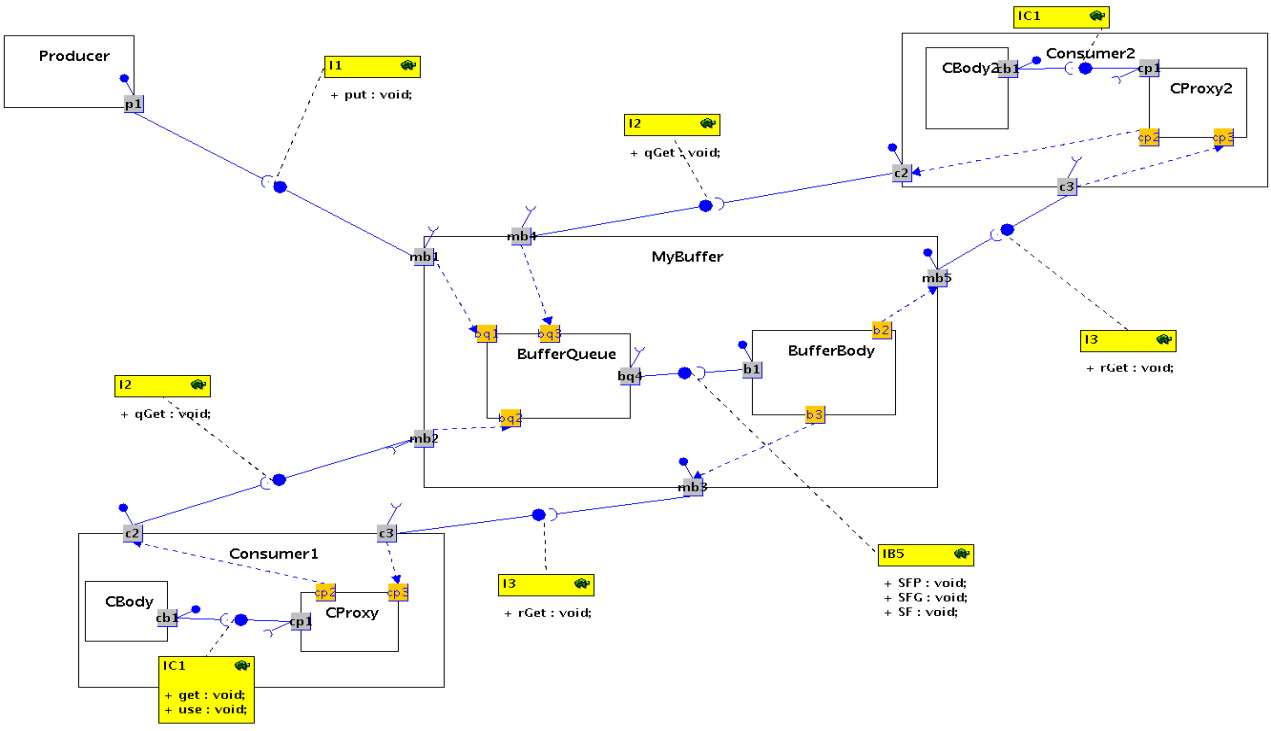
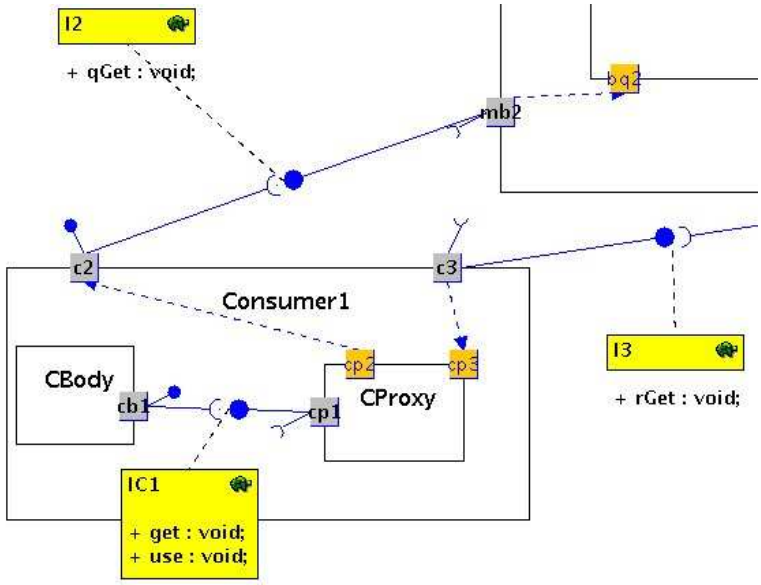


Figure 5.2.2 Composite Structure Diagram of Consumer-Producer system with CTTool

We will describe each component as it follows**:**



A consumer is composed by the primitives C**Body** and C**Proxy**.

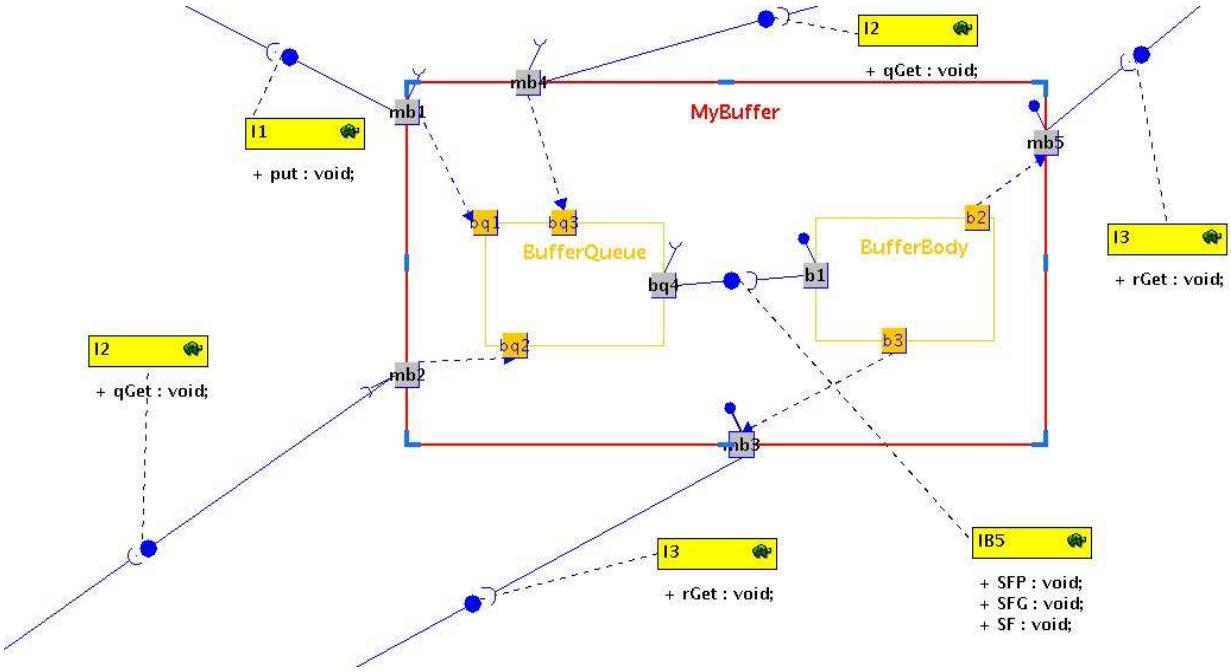
an Interface (IC1) defines the messages traveling between the two primitives.

The body will send a "get" message (via cb1 port) to the proxy and then it will continue it's execution.

In a later moment, when it needs the value it will try to use it and will block if the value is not yet arrived.

The Proxy receives the messages from the body via port cp1. It will send a "qGet" message via delegate port cp2. This message will pass through the father's port c2 to go to the buffer.

Proxy will now wait a response (rGet) from the buffer in the cp3 port. When this message arrives, the proxy will allow the Body to use the value.



MyBuffer Components is composed by BufferQueue and BufferBody

BufferQueue store requests (put and get) from consumers and producers while BufferBody is the engine who takes the requests from the queue and process them. This approach to the standard behavior of a ProActive object.

The message transit description:

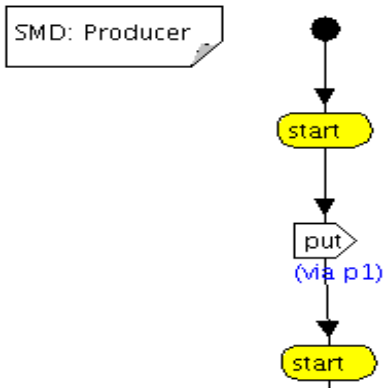
- "put" message from the Produce gets into the Queue via mb1 port and bq1 delegate port.
- "qGet" message arrives from the Consumer1 into port mb2 and then into delegate port bq2.
- the same "qGet" message from the Consumer2 will arrive into the port mb4 and delegate port bq3.

All this messages will be stocked into the Queue waiting for the BufferBody to treat them.

The buffer body will "pick up" requests from the BufferQueue by calling methods defined in the interface IB5 (ServeFirstPut, ServeFirstGet, ServeFirst) through the port b1. It will treat those requests and send responses to the consumers through the ports b3->mb3 and b2->mb5.

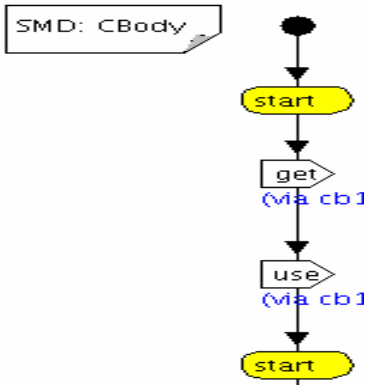
We will have to describe behavior of the primitive components (BufferQueue, BufferBody, CProxy, CBody,Producer) by designing the

5.2.2 State Machines Diagrams with CTTool



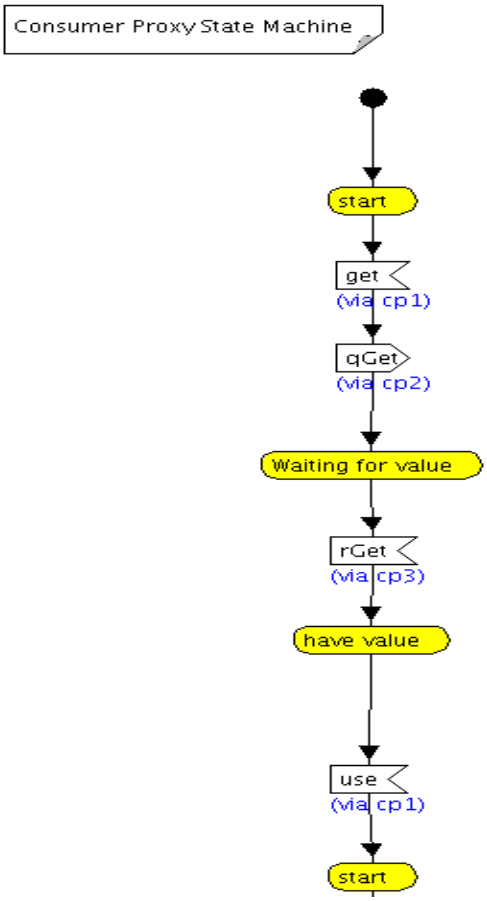
Producer is in the start state.

It will send a "put" message via port p1 and will move again in the start state.



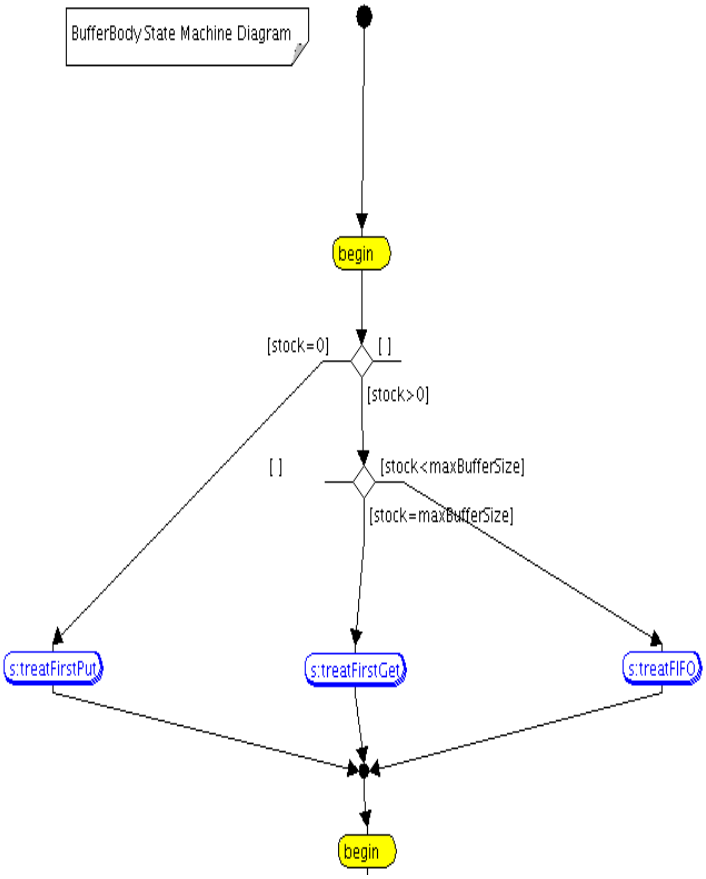
In our present model the consumer will send a get message via cb1 port.

Then it will try to use the value and call use method via cb1 port. At this point, if the proxy doesn't yet have the value, the CBody will block until the value becomes available.



The consumer Proxy it is initially in the "start" state. It will receive a "get" message via cp1 (from the Body). It will send a "qGet" message via cp2 (a request to the Buffer) and put itself in "Waiting for value" state. After receiving "rGet" message via p3 (a response from the buffer) it pass in "have value" state and allows the consumer to call "use" method via cp1.

After the value is used it restarts the cycle from "start" state.



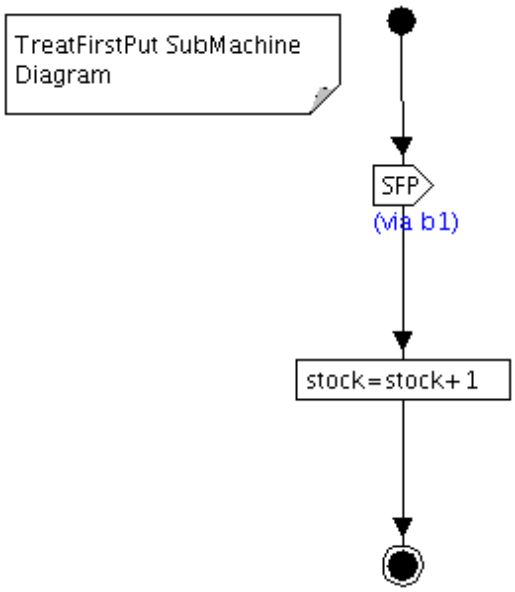
The BufferBody choose one of the treatment strategies depending on the actual stock value.

Three Sub Machines are defined.

If the stock is empty it treats the first put request in the queue (because it can not treat get requests, it doesn't have any information in the stock).

If the stock is full it only treats getRequests into the queue (it doesn't have any more free space to store information in the stock)

If the stock is not full nor empty it treats in FIFO order.

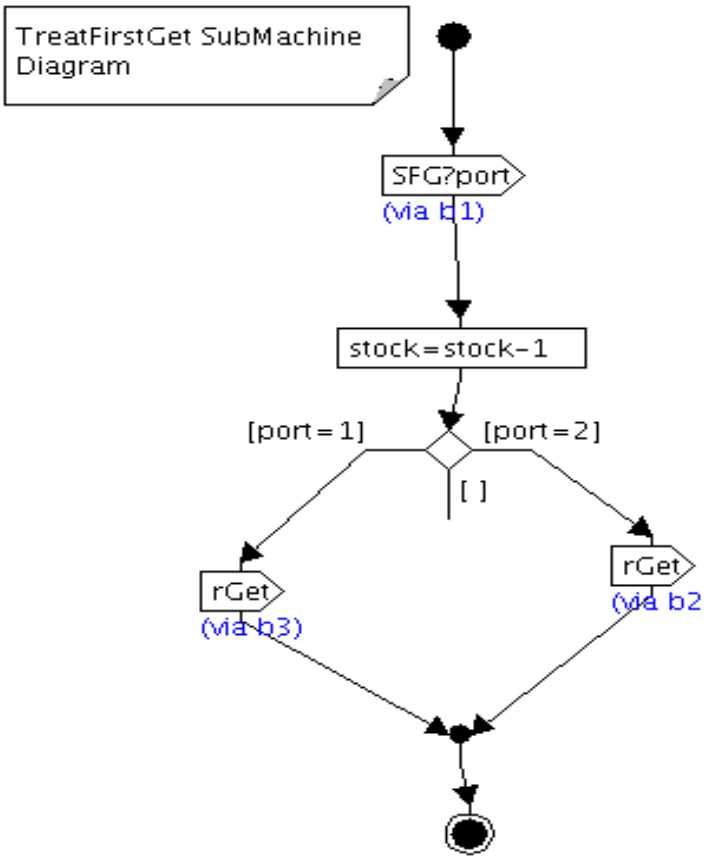


TreatFirstPut Sub-Machine:

The buffer pickup the first "put" request in the Queue by calling SFP (ServeFirstPut) method via b1 port. The stock increase with one unit.

Then the Sub-Machine ends and the calling machine will continue with the first operator after the Sub-Machine.

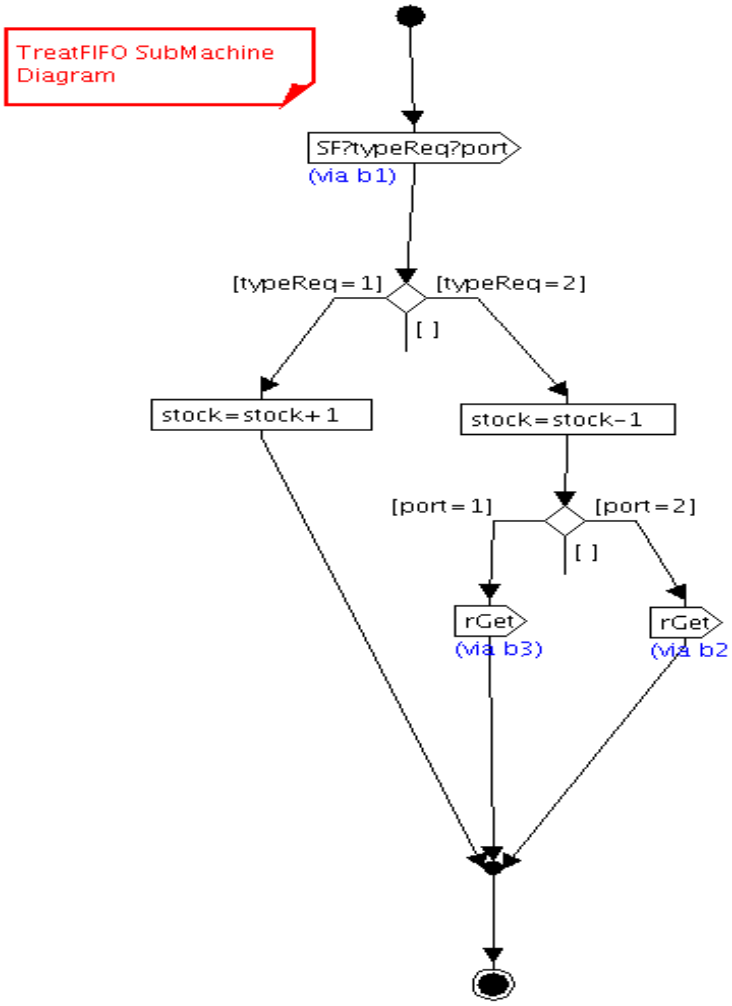
Remark: We don't need to model the values (the information) stored into the stock. That would enormously increase the size of the final reachability graph without changing in any way the system behavior.



The stock is full so the buffer will pick up the first "get" request into the Queue by calling "SFG" method on port b1 and receive "port" parameter. The "port" identifies the consumer who sent this requests and who will receive the answer.

The stock decrease one unit.

Then, depending on the received parameter "port", the "rGet" message will be sent via b3 port (to Consumer1) or via b2 port (to Consumer2) - see Component Diagram)

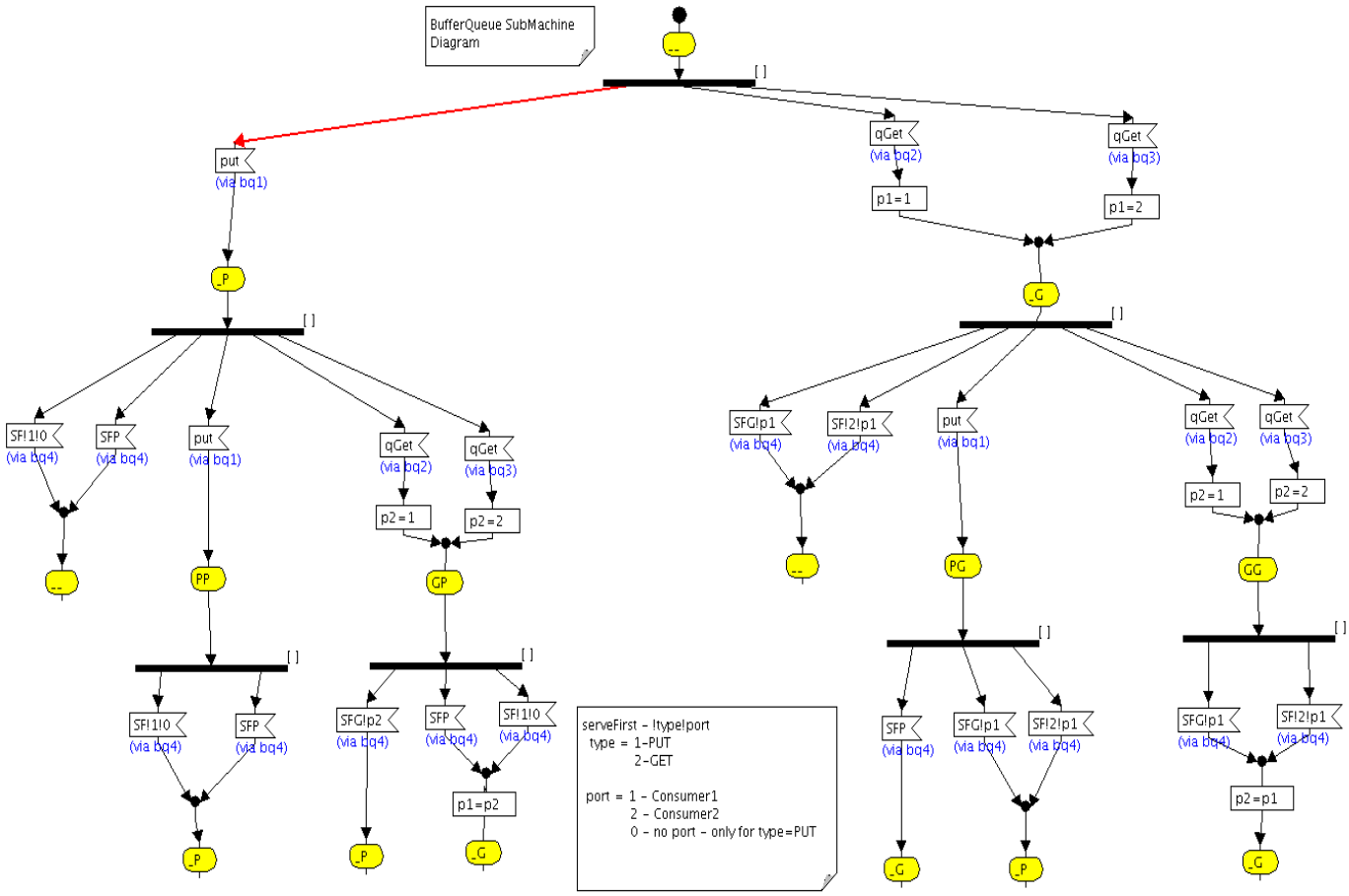


FIFO Sub-Machine: Buffer will pick up the first request from the Queue and treat it.

The buffer calls an SF(Serve First) method via b1 port and receives typeRequests (which could be 1 for PUT or 2 for GET) and port to identify, in case of a request of type "get", the owner (sender) of the request.

In case of a put request (typeReq=1) the stock will be increased.

In case of a get request (typeReq=2) the answer ("rGet") will be sent through the port corresponding to the sender of the request.



BufferQueue State Machine Diagram

We have modeled a Queue of length 2.

- "_ _" state: the queue is empty - no requests are in the queue. It can receive "put" message or "qGet" message via two different ports, one corresponding to each Consumer. The p1 attribute will store the port corresponding to the Consumer who sent the request on the first position in the Queue (from right to left). The p2 will do the same for the second position in the queue.

- "_P" state: a put request is on the first position in the Queue. The second position is empty.

It can receive SF message (from the body) and send back !1!0 values meaning that the type of the first request in the queue is 1 (put) and the port corresponding to the owner is not important (is 0) because the Producer doesn't wait for any response so we are not interested in who the producer is.

It can also receive SFP message (from the Body) , put message from the Producer and "qGet" message from each of the consumers.

We let the reader of this paper to further look this diagram to understand how the BufferQueue works in each case. The example is also available for download.

5.2.3 Generation and Checking of Formal Code

We can now analyze the diagrams, generate Lotos Specification, Check Syntax of formal code and generate the Reachability Graph.

Formal code generation:

- click the 'Syntax Analysis' icon on the main bar

  - This only checks some properties of your design before generating the Lotos specification, the final checking is done on the Lotos code, by Caesar.

- click on 'Generate Lotos Specification' icon

- click on 'Check Syntax of Formal code' icon

  Your Lotos file 'spec.lot' is generated and checked for errors. You can find it in Turtle/bin directory or in the main menu 'View-> Show last formal specification'.

From this point model checkers can be used with the Lotos specification as input. For doing that, model checkers can be directly used or used through Turtle Menus.

Example:

Generation of the Reachability Graph:

- click on 'Run validation' icon, choose your options and push the Start button. The reachability graph is generated and located in 'Turtle/bin/spec.bcg'.

See [5] (http://labsoc.comelec.enst.fr/turtle/HELP -> formal validation) for more information.

For this example we obtain a Graph of 577 states and 1013 transitions. We can use V&V menus to analyze the Graph.

Use 'View' and 'V&V' menus to visualize and analyze the generated graph.

Note: by analyzing the generated graph we have found few deadlocks. An execution that leads to a deadlock would be the next one:

1. Producer put.
2. BufferBody serve put
3. (1,2) two times
4. Producer put - 3 times
5. Consumer get
6. **Deadlock**

At this moment (step 6):

- the stock is full(length=3).
- in the buffer's waiting queue are 3 producer's requests (put information) which makes the queue full.
- a consumer is trying to send it's request to the queue
- the buffer doesn't have space into the queue to introduce the consumer's request
- the buffer cannot serve any producer's requests because it's stock is full.

A solution would be to anticipate this situation and not to introduce the last's producer requests into the queue.

5.3 CTTool reference manual of alpha version

Build as an extension of Ttool, CTTool uses several facilities offered by this one. As well, most of types defined in the new packages inherit already-defined types from Ttool.
As we have seen in the previous chapter, our extension proposes the possibility of a system design based on UML2, more specific, Component Diagrams (Composite Structure Diagram), to describe system architecture and State Machine Diagrams to describe system behavior.
The program offers the user a graphical interface for drawing those two types of diagrams, and is able to parse them and create a TurtleModeling object. We use then the engine of TTool to manage this object (model) and create a formal specification (Lotos code).
We will explain in what follows the conception of the graphical interface and the engine that parse it and creates the TurtleModeling.

5.3.1. CTTool graphical interface

Composite Structure Diagram

Graphical objects are stored on a ProactiveCSDPanel.
We have three types of graphical objects: those who inherit TGComponent, those who inherit TGConnector and those who inherit TGConnectingPoint.
Some of the Graphical Components dispose of a set of ConnectingPoints.
Each Connector has references of two ConnectingPoints connecting in this way the owners of the ConnectingPoints. It can also have a direction, from the first ConnectingPoint to the second.

Few specifics of each type of element on the user design graphical interface.

ProCSDComponent extends TGCWithInternalComponent implements

SwallowTGComponent, SwallowedTGComponent, ActionListener

A component may have it's internal components. In this case we call this component a father and the others sons. A father movement will imply a movement of each of it's sons. On the other hand, a son will only move within the father area.

A component has a set of attributes which can be defined and initialized by the user.

On double-click the State Machine corresponding to this Component is created if it doesn't already exists and it's panel becomes the active panel.
We will see in the next chapter that the State Machines of “father” Components called Composites are ignored and only the State Machines of Primitives are included in the model.

ProCSDPort.

Three kinds of port have been implemented, ProCSDInPort, ProCSDOutPort, and ProCSDDelegatePort, all inheriting ProCSDPort.

A port can be connected to another with a connector or a delegate connector via the connecting points.

Connector (TGConnectorProCSD) and **Delegate Connector** (TGConnectorDelegateProCSD)

A Connector connects an OutPort P1 with an InPort P2. An interface is associated to the connector. That means that all the messages defined in the interface (and only them) can transit form P1 to P2. This will be a restriction in the State Machine Diagram.
A delegate connector can connect:

An in port P1 to a delegate port P2 : messages coming to a Composite via port 1 are sent directly to one of it's sons via port P2. P2 is delegate to receive those messages.

A delegate port P1 to an out port P2: messages created or received by a son of a composite C are sent from the port P1 of the son to the port P2 of the father to be sent forward outside the Composite.

A delegate port P1 of a Component C1 to a delegate port P2 of a component C2: when C2 is a son of C1 and C1 is itself a son of a composite C. Or vice versa. A message received by a composite can transit via several delegate ports in the hierarchy to a subcomponent who will treat the message. The same in the opposite way.

ProCSDInterface

An interface is associated to a Connector (TGConnectorProCSD). It defines messages (methods calls) that pass through this connection. As the connection has a direction, let's say from port P1 of C1 to port P2 of C2, it means that the messages transit from C1 to C2. We can see this as methods called by C1 on C2. Messages can be of type void or may have a return type.

In this alpha version of CTTool we create a connection between two ports (which implies a connections between two components) and associate an interface to this connection.
In the beta version we will let each primitive component specify the interfaces it offers and those it requires and then just bind components.

State Machine Diagram User Design

For each primitive component the user have to specify it's behavior by designing a State Machine Diagram.

The next operators compose a state machine:

- a start state and a stop state
- a state operator
- condition operator
- sendMessage and getMessage operators

  For each of this two operators a port must be specified to indicate where the messages come from or go to.
- action operator: an action on a local attribute
- nondeterministic choice operator
- submachine operator

The operators offered by this diagram will be detailed in the next section on the creation of the TurtleModeling.

5.3.2 Parsing diagrams and creating the TurtleModeling

We have implemented a class (GProactiveDesign) who parse the diagrams in order to create the TurtleModeling

It first analyzes the class structure, determine who are the primitives, the connections between them, what is the transit of messages, what is the behavior for each component and put this information into a TurtleModeling object.

The Composite Structure Diagram will be mapped on a Class Diagram and the state Machine Diagram on an Activity Diagram as follows.

We will create a class for each primitive component, add a public gate for each pair (port P, message M transiting through P). Synchronizations between classes will be added. An activity diagram will be created from the Sate Machine Diagram of each primitive.

In the first step, to each in/out port in the CSD, we have to add this information:

- the interface who specifies messages (methods) that transit to this port
- the complementary port connected to this one.(i.e. For an out port P1 of a primitive C1 find the in port P2 of a primitive C2 where the messages from P1 will finally arrive.)

Each port has the local attributes:

- my Interface – the Interface who specifies messages that transit through this port
- toPort – the port where a message going out from this port will arrive.
- fromPort – the port where a message getting in this port is coming from.

Update Ports Information Method

- for each out port outPort (outPort of C) on the CSD set the toPort.
  toPort is the in port directly connected to outPort.
  toPort.fromPort is outPort.
  - Then we iterate a chain of connected delegate ports p1 ..pn of the components C1 ... Cn with the propriety:
    C1 is a son of C and p1 is connected to outPort
    C(i) is a son of C(i-1) and p(i) connected to p(i-1)

- for each of the ports Ci we update toPort information and for toPort we update fromPort information. End for.

- end for

We do the same for inPorts.

More formally, the algorithm for updating ports information is:

Update ports information Methode

- For each out Port pOut (of a component C) on the diagram do:
  - find and update pOut.myInterface
  - if interface not found create a CheckingError and return
  - port PIn = the in port directly connected to pOut
  - pOut.toPort=PIn
  - PIn.fromPort=POut
  - DP=the delegate port of a subcomponent of C connected to pOut (wich means the messages come from PD, transit through POut to outside the component C)
  - if DP not null
    - DP.myInterface=pOut.myInterface
    - pOUT.fromPort=DP
    - DP.toPort=pOut
    - DDP = DP.getFromInsideConnectedPort
    - while DDP not null
      - DDP.myInterface= pOut.myInterface;
      - DDP.ToPort=pIn;
      - pIn.FromPort=DDP;

- DDP=getFromInsideConnectedPort(DDP);
- end while
- end if
- end for
- for each in Port inP (of a component C) on the diagram do
  - look for and update inP.myInterface
  - DP=the delegate port of a subcomponent of C connected to pIn (a port that pIn delegates to treat messeges pIn receives)
  - if DP not null
    - From Port = pIn.fromPort
    - DP.myInterface=pIn.myInterface
    - DP.fromPort=fromPort
    - fromPort.setToport(DP)
    - ddp=getToInsideConnectedPort(DP);
    - while (ddp!=null)
      - ddp.setMyInterface(myInterface);
      - ddp.setFromPort(fromPort);
      - fromPort.setToPort(ddp);
      - ddp=getToInsideConnectedPort(ddp);
    - end while
  - end if
- end for
- end updatePortsInformation

Creating the TurtleModeling

Method addTClasses

- For each primitive C :
  - we create a TClass TC who corresponds to that primitive
  - we add the attributes of the component to he TClass
  - for each port p of C
    - for each methode m defined in p.myInterface
      - create a gate p_m and add this gate to the TClass.
    - end for
  - end for
  - build activity diagram of C
- end for
- add all synchronizations and the synchronization gates between TClasses.
- end method addTClasses

Building the Activity Diagram :

Each operator on the State Machine is mapped on an operator from Activity Diagram like in 4.3 page 21.

SendMessage and GetMessage operators are transformed into the correspondent synchronizations.

A submachine operator will be replaced by the submachine itself.

The TurtleModeling being generated, the TTool engine can take it as input to generate the Lotos code.

6. Conclusions and further work

6.1 Evaluation

We have designed and analyzed in this report two different models for the Consumer-Producer example, first model based on the Classical TTool with Class Diagrams and Activity Diagrams, the second on the UML2 Component Diagrams and State machines diagrams.
A comparison between the two specifications led to the identification of our requirements for a new Tool.

We have design and implemented the basis for a new design tool that we called CTTool. In order to do that we have specified the elements that compose the two new diagrams: Composite Structure Diagram and State Machine Diagram.

As we wanted to use the already-implemented Turtle engine for the generation of Lotos code we mapped the UML2-based on model to the Classical TTool model. This means adding well defined semantics to each element of the new diagrams.

We have written the algorithms needed to parse the new defined diagrams, collect information and create the model that is given as input to the Turtle engine in order to generate Lotos code.

These algorithms have been implemented and led to a first prototype of our new Design Tool, CTTool. This prototype has a friendly user-interface and semantics defined for all the provided modeling operators.

We have also offered a complete specification of the Consumer-Producer System within the new implemented Tool, CTTool. We have generated the formal code for the system and used Caesar (from CADP) to create the accessibility graph. A study of this graph led us to discover few deadlocks in the model of our Consumer-Producer.

We have seen that the translation we have made (when parsing diagrams and creating the model) implies some significant information loss.

As we do not intent to transform a component-based system in a simple object-based system (which would mean going backwards) we intent, in our future work, to keep this information (that we loose during the translation) in the turtle model.

There are still few features that need to be added before offering CTTool to the real-world system developers. We present these features in the next section.

6.2 Further Work

We will need to design and implement a beta version of CTTool, as the alpha version we have presented in the Chapter 5 is only meant for the research aria.

As the beta version is meant for designing component-based system, we also intent to develop a gamma version specialized in modeling ProActive systems.

Several features that would have to be added or changed in the current version of CTTool in order to distribute it to the users:

Features to be improved in the beta version

- design of a component with its provided and required interfaces
- allow user to bind only the mandatory interfaces
- allow the user to create libraries with his defined components and their behavior
- define constraints that allow replacing a component in its context
- a better management of errors

Features to be improved in the gamma version:

- design and implement the model of a standard ProActive Object (Component) with the behavior of its standard subcomponents the user could use on his diagrams
- manage multiplicity: instantiation and binding of multiple instances of a component
- group communication
- generation of pNets models
- manage simple-types lists

Even if CTTool alpha version is only meant for research purposes, it approaches of a Component System Design Tool and opens the perspective of a professional Proactive Design Tool.

7. References

[1] BARROS Tomas

"Formal Specification and Verification of Distributed Systems" (Spécification et Vérification formelles des Systèmes de Composants Répartis)". Thèse de doctorat, , Université de Nice - Sophia Antipolis, INRIA-Laboratoire I3S, 25 Novembre 2005, Projet OASIS.

[2] BARROS Tomas, Rabea Boulifa, Eric Madelaine

Parameterized Models for Distributed Java Objects, Article , INRIA Sophia Antipolis

[3] Object Management Group (OMG)

Unified Modeling Language: Superstructure, version 2.0, Revised Final Adopted specification (FTF convenience document), ptc/04-05-02, http://www.omg.org/uml/

[4] Matěj Polák

UML 2.0 Components, Master's thesis(Oct 2004 - Sep 2005), Distributed Systems Research Group, Charles University, Prague
http://nenya.ms.mff.cuni.cz/~mencl/projects/uml20components-thesis.html

[5] Ludovic Apvrille

Turtle Documentation http://labsoc.comelec.enst.fr/turtle/HELP/

[6] L. Apvrille, P. de Saqui-Sannes, J.-P. Courtiat

A UML 2.0-based Evolution of the TURTLE Profile, Article, ENST Institut Eurecom

[7] Rumbaugh, James,Jacobson, Ivar, Booch, Grady

The Unified Modeling Language reference manual, Campus press , 2005

[8] Rabea Ameor-Boulfia

Génération de modèles comportamentaux des applications réparties,
Thèse de doctorat, , Université de Nice - Sophia Antipolis, INRIA-Laboratoire I3S

[9] ProActive, Oasis Team, www-sop.inria.fr/oasis/ProActive