Available online at www.sciencedirect.com
SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science

ELSEVIER Electronic Notes in Theoretical Computer Science 130 (2005) 187–209
www.elsevier.com/locate/entcs

Protocol Performance Analysis Using a Timed Extension for an Object Oriented Petri Net Language[1]

Fabrício Vale de Azevedo Guerra[2]
Jorge Cesar Abrantes de Figueiredo[3]
Dalton Dario Serey Guerrero[4]

Universidade Federal de Campina Grande
Departamento de Sistemas e Computação
Coordenação de Pós-graduação em Informática
Av. Aprígio Veloso, s/n – 58109-970 PB
Campina Grande, Brasil

Abstract

In this paper we propose a timed extension for RPOO, a Petri-net based object-oriented modeling language. The timing strategy of the proposed extension, called RPOOt, is based on the timing strategy from the Timed Coloured Petri nets formalism. A simple stop-and-wait protocol RPOO model was developed in order to demostrate the proposed extension. We also show some performance measures for the model. RPOOt showed to be easily integrated with Timed Coloured Petri Nets, providing means for modeling time consuming activities *inside* objects as well as *among* objects.

Keywords: Timed Petri nets, performance models, object orientation.

[1] This is an extended version of the article published at the *Brazilian Symposium on Formal Methods (2004)* under the title *Timed Extension for an Object Oriented Petri Net Language.* This work is supported by CNPq Brazilian Research Council, MOBILE Project, process 552190/2002-0. First author supported by CAPES.
[2] Email: `guerra@dsc.ufcg.edu.br`
[3] Email: `abrantes@dsc.ufcg.edu.br`
[4] Email: `dalton@dsc.ufcg.edu.br`

doi:10.1016/j.entcs.2005.03.011

1 Introduction

RPOO [8] is a Petri-net based object-oriented modeling language. It can be seen either as an object-oriented extension for high-level Petri nets or as a way to give formal concurrent semantics for object-oriented models. In practice, it allows object-oriented decomposition of complex high-level Petri nets models. The main difference between RPOO and other object-oriented extensions for Petri nets is that RPOO stresses semantic composition instead of syntactical integration. Based on this idea, composition of smaller Petri nets is achieved by means of object-oriented semantic mechanisms, instead of the conventional hierarchical ones [8]. These characteristics make RPOO easier to use in object-oriented real world projects.

RPOO has been successfully applied to the modeling of several systems. Some of these applications had *academic* purposes [8,15] whereas others aimed at *testing* the formalism by using it to model, simulate and/or verify real world systems [4,5]. A comprehensive case study was conducted in order to model and analyze the Mobile IP protocol [13], a standard that provides mechanisms for keeping connectivity of mobile hosts that migrate through different networks.

RPOO proved to be suitable for the modeling and analysis of communication protocols, including those with mobility features. However, it lacks explicit mechanisms to cope with time. The first evidence of the need for a timed extension occurred during the modeling of the Mobile IP protocol [5]. Many of the protocol features included time constraints: for instance, resending a request after a *timeout* has elapsed. During the analysis of the protocol, some timing properties were verified. For example: in order to stay connected after migrations, one of the operations a mobile host must perform is to obtain an IP address local to each network it *visits*. For a number of security reasons, the protocol defines that this address must be valid only in a given frame of time. RPOO models can easily address the fact that this local address must not be hold indefinitely, but they cannot do the same about mobile nodes keeping the address *for* a specific amount of time.

Moreover, in such a sort of protocols like the Mobile IP, there is a lot of interest in measuring performance considering different scenarios. For instance, it would be necessary to evaluate data loss during migration. Since it is not possible to explicitly express timing aspects in the RPOO models, it is not straightforward to perform time constrained functionalities analysis.

This way, we could not express in our RPOO models for Mobile IP such timed features and its validation did not include performance measures and time constrained functionalities analysis.

In order to model (and analyze) these functionalities and measure performance, it is necessary to create extra classes (nets) to cope with time. *All* the *actions* including time parameters should then be connected with these classes. Changing time during simulations requires a complete synchronization between the classes created to model time and all the remaining classes. This is necessary to guarantee that there is no possible action to be executed in the current time. With these modeling problems, analysis would be difficult and error prone, since the RPOO modeler should also *validate* his time strategy. Thus, we can say that, *in practice*, RPOO language *cannot* model this kind of feature.

A *timed* extension is necessary to properly deal with these problems. Since RPOO models have both the OO and the Petri nets perspectives, the definition of such extension must consider timing schemes for each one of them.

Over the last decade many valuable timed OO models were proposed, many of them extending UML. UML-RT, as presented by Kruger [11], brings some real-time modeling related features from the ROOM language [16] to UML in order to make it more suitable for the modeling of systems having time constrained functionalities. Shu gives formal semantics to UML statecharts and formally verifies the models by means of timed automata [17]. Real-time UML is another extension of UML for real-time modeling and has the Rhapsody tool to support it [2]. Experiences with Real-time Corba [12] were also proposed and a number of other “non-UML” extensions [1,18] may be found in the literature.

Concerning the Petri nets world, many different models with time have been defined since the seventies. Many authors proposed to augment the model by considering delays and time intervals attached to Petri net structure elements like transitions, places and arcs [6]. Most of these models have proved to be equivalent thought some authors claim that there are some non-equivalent classes [3]. The earliest models were proposed over Place/Transition nets. However, some authors also defined temporization for High-level Petri nets. Jensen [10] defined the Timed Coloured Petri Nets (TCP-nets), a coloured Petri net extension where *typed* tokens carry timestamps.

Most of the timed OO models were conceived for the modeling of real-time systems, which present a considerable degree of concurrency and parallelism. It seems that a lot of effort was focused on giving OO models a semantic for coping with such features. That justifies the creation of elements like capsules, bindings and ports in languages like ROOM and UML-RT. As a result, time treatment, in most of the models, is reduced to the association of fixed delays between states in statecharts that describe objects *internal* activities (this is not a good strategy to model time consuming interactions *between* objects).

For Petri nets, the researchers focus on temporization. The original formalism was already suitable to deal with concurrency and parallelism. This way, more accurate strategies were proposed in the Petri nets world along three decades of research on the subject.

The main objective of this work is to present an RPOO timing extension to explicitly cope with time in RPOO models. In practice, it means that the RPOO formalism can be used to model real-time systems as well as to make performance evaluation. The central idea is to define a timing OO-level strategy that can be used with the accurate timed models that are already defined for Petri nets. In this way, the OO models can inherit some of the clever temporization aspects from the Petri nets world. As a gain, already suggested by practical experiments [5], a RPOO extension is able to provide time constrained functionality modeling and performance measure/analysis.

Currently, RPOO uses Coloured Petri nets (CP-nets [9]) for describing its classes. The proposed extension aims at integrating the formalism with the TCP-nets, in which tokens are timestamped and the models have a global clock. For that purpose, a global clock was added to RPOO models, all the messages exchanged between objects were timestamped and some transition rules redefined, taking both clock and timestamps under consideration. Thus, the proposed timed extension to RPOO may also be seen as an *OO extension* to TCP-nets.

The rest of this paper is organized as follows: In Section 2 both RPOO and RPOOt are informally introduced. A case study is presented in Section 3. The analysis results from simulation of the case study models are shown in Section 4. In Section 5 RPOO is formally defined together with its main semantics. The proposed timing extension is defined in Section 6. Section 7 concludes the paper.

It is assumed that the reader has some elementary skills on timed Petri nets and is familiar to basic OO concepts and syntax.

2 Informal Overview of RPOOt

RPOO models consist of a set of classes and their corresponding Petri nets. Classes are described and can be related to each other just like UML classes. The Petri nets describe the behavior of objects—there is exactly one net for each class in the model. For this reason, an RPOO model can be obtained from a class diagram. For each class, we have a CP-net to describe it. A variety of *RPOO actions* (instantiate objects, call methods, destroy objects...) can be performed by the objects and actions are depicted by transition *inscriptions* in the CP-nets. An inscription may describe several actions and all the actions in an inscription are executed atomically. A set of atomically executed actions is called an *event*.

In RPOO each object is a *thread* and interaction between two objects may be *asynchronous*. This means that when an object *a* calls a method of (sends a message to) object *b* in asynchronous mode, the system moves to a state where the data passed as parameter will be *pending* and may be *consumed* in a further action by object *b*. RPOO actions also include *synchronous* calls, where messages are sent *and* consumed atomically.

A set of interconnected objects and its *pending messages* form a *structure* of an *Object System*. Besides this structure, an Object System also knows what we call *imminent* actions, that is, actions that may be executed in the current structure. Briefly, an RPOO Simulator keeps track of an Object System, executing actions and deciding which action will be executed, in case of concurrency.

Figure 1 illustrates a configuration for an object system with two objects. The Petri net details that describe the behavior of each object are abstracted. Only the communicating transitions (graphically represented by rectangles) together with their corresponding inscriptions are shown. For visualization purpose, the thicker transition means that it can fire and its RPOO inscription (**object2.set(data)**) will be executed. This inscription denotes an asynchronous method call.

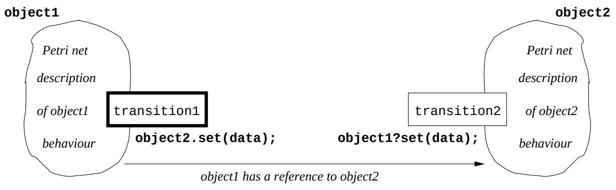
object1
Petri net description of object1 behaviour
transition1
object2.set(data);
object2
Petri net description of object2 behaviour
transition2
object1?set(data);
object1 has a reference to object2

Fig. 1. 'object1' is ready to send a message to 'object2'

After the execution of the asynchronous call, the new resulting configuration is illustrated in Figure 2. It shows a *pending* message (**data**) from **object1** to **object2**. It also shows that **transition2** in **object2** is *enabled* and may execute an *input* action, that will *consume* the pending message (**data**). This kind of action is denoted by the **?** symbol. The inscription **object1?set(data)** means that, in order to *fire* this transition, **object1** must have called the **object2.set(data)** method in an *earlier* action (note that the concerned transition must also be *enabled*).

Figure 3 shows the state of the system after the execution of the input action of Figure 2. Note that there is no longer pending message and there are no transitions enabled.

Differently from RPOO, RPOOt models have a global clock (the **model**

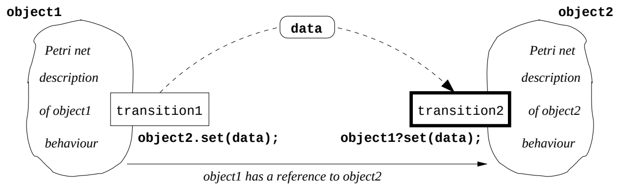

Fig. 2. A *pending* message from 'object1' to 'object2'

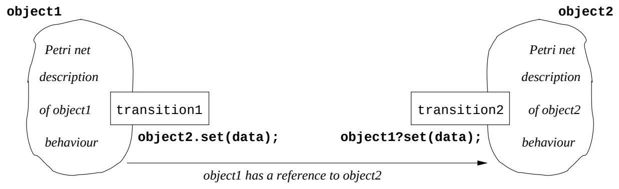

Fig. 3. Message *consumed* by 'object2'

time) and the messages may be *timestamped*. When an asynchronous method call is executed, the resulting pending message may have a timestamp that indicates the least model time at which the message can be *consumed* by an appropriate input action. *Ready* messages are those which have timestamps less than or equal to the model time. *Only* ready messages can be consumed by input actions. When no more actions can be executed at the current time, the model time must be set to the least value at which an action can be performed. Thus, a message having a timestamp t units greater than the model time takes, at least, t time units to be consumed. It means that the concerned action *takes* at least t time units to be executed.

Figure 4 gives an informal view of a *timed* configuration for an object system with two objects. The current model time is **10**. Again, the Petri nets that describe the objects behavior are abstracted. The thicker rectangle means that the abstracted Petri net gives the represented transition conditions to execute its RPOOt inscription (**object2.set(data)@+5**). The semantic of **@+5** is that the **data** sent to **object2** will have a timestamp 5 units of time greater than the current model time (**10**). This inscription denotes a *timed* asynchronous call of method.

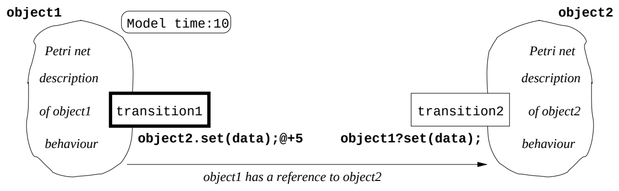

Fig. 4. 'object1' is ready to send a *timed* message to 'object2'

After the execution of the **object2.set(data);@+5** action by **object1**, the resulting configuration is illustrated in Figure 5. It shows a *pending* timestamped message (**data@15**) from **object1** to **object2**. It also shows that **transition2** in **object2** is not *enabled* since the timestamp of its input message (**15**) consists of a value greater than the current **model time** (**10**).

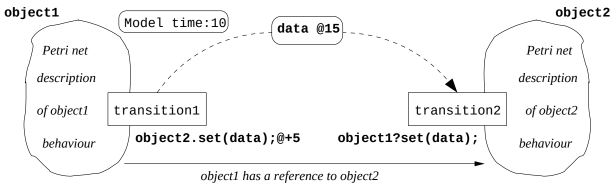

Fig. 5. A *pending* timestamped message from 'object1' to 'object2'

Since there is no more imminent actions, the object system **model time** is set to the least value in which an action is imminent. This operation will result in the configuration shown in Figure 6. Then, with **model time 15**, the message having **data** as contents is *ready* to be consumed by the input action **object1?set(data)** in **transition2** (from **object2**). The effects of *consuming* the message in this configuration are similar to those shown in Figure 3 (except that now the resulting configuration still have a **model time**).

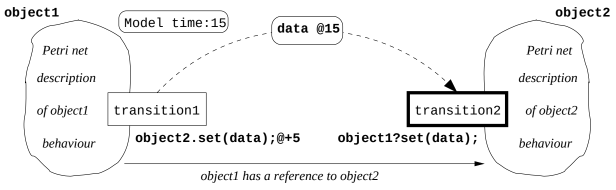

Fig. 6. A *ready* message from 'object1' to 'object2'

3 Modeling Timed Protocols with RPOOt

In previous sections, it was shown an overview of how RPOO and RPOOt models operate, focusing on configurations of object systems that concern internal aspects of an RPOO Simulator. In this section, it is presented the *developer view* of RPOOt. A simple stop-and-wait protocol over an unreliable network is used as a case study. Since the main objective is to use TCP-nets to describe the objects behavior, the Petri net models that represent the objects internal behavior in this case study are strongly influenced by the models described by Jensen [10].

The protocol must guarantee that packets transmitted from a server to a client arrive in the correct sequence. A sequence number scheme is used together with a positive acknowledgment strategy. To accomplish this, the client must send an acknowledgment to the server each time a data packet arrives. When an ack message arrives at the server, it sends the next packet of the pre-defined sequence and so on. Since the network may loose packets, the server must resend copies of the same packet every time a delay has elapsed without receiving a corresponding ack message.

The protocol does not consider the way the server breaks the original message in several sequenced packets neither the way both client and server agreed about the sequence number of the first packet to be sent. The protocol concentrates on sequencing. Informally speaking, the protocol must respect the following properties:

- Any packet having a sequence number n is always received at the client side before the packet with sequence number $n + 1$.
- Possibly, all the packets from the original sequence will arrive at the client side.

Since transmition over the network is not reliable, it is not possible to guarantee that all of packets will arrive at the destination side. That is the reason why a timed model along with formal reachability verification is usefull. By checking the model it is possible to assure that all the packets will *possibly* arrive at the destination and that the sequencing is preserved regardless the number of correctly transmitted packets. Through performance evaluation, developers can see wether the conditions in which the system operates are acceptable, avoiding the necessity to model time mechanisms explicitly.

3.1 The RPOOt Model

An overall picture of the model is represented by the class diagram in Figure 7. A **Network** object may be connected to several **Nodes**, which can be either a **Server** or a **Client** object. Each node, in turn, has a reference to a **Network** object, so it can communicate with other **Nodes**.

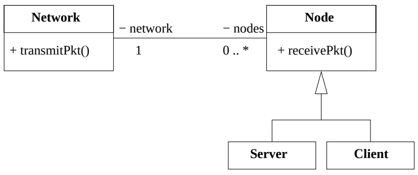

Fig. 7. Class Diagram for the Stop-and-wait Protocol

The sequence diagram of Figure 8 shows an ideal scenario for the protocol, without packet loss. It considers the object **server** communicating with the object **client** through the object **network**. Essentially, the **server** addresses a data packet to the **client** and invokes **network.transmitPkt()**

passing the concerned data packet as a parameter. The **network** then invoke **client.receivePkt()**, delivering the packet to its destination (again, the packet is passed as a parameter to the considered method). A similar procedure occurs in order for the **client** to send an acknowledgment message to the **server**.

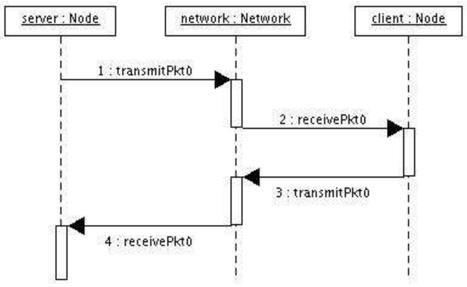

Fig. 8. Sequence Diagram for the Stop-and-wait Protocol

In what follows, some nets of the stop-and-wait RPOOt model are shown together with a brief description of their behavior. Note that it is not possible to identify in the RPOOt classes the *configuration* of the *object system* and the *model time*: it is up to the simulator to keep track of object links, bind variables and perform the *actions* inscribed in the nets.

In the Server model (Figure 9), there are two transitions and three places. It consists of a TCP-net with RPOOt inscriptions, modeling timed activities internal to Server objects as well as time consuming interactions between Server objects and Network objects. The initial marking is depicted in the figure by the inscription on the top of places. For instance, the initial marking of place **NextSend** is defined by inscription **1'(Client1,1)++1'(Client2,1)**. So, there are two tokens in place **NextSend**: one token of value **1'(Client1,1)** and one token of value **1'(Client2,1)** (the operator ++ represents multi-set addiction, as used in CP-net models). This initial marking indicates that the next packets to be sent to both clients have sequence number equals to **1**. The **DataToSend** place initially holds all the packets that a Server object must send and the **NextSend** place controls the number of the next packet to be sent (for each destination client). A **PACKET** token has references to the **sender** and destination (**dest**) objects and a *content*, which can be, in this implementation, a data (**Data**) or an acknowledgment (**Ack**) message.

When the **SendPacket** transition fires, it takes a token from the **DataToSend** place. This token is used as parameter to the RPOOt message **network.transmitPkt(sender, dest, Data(n,d));@+Tsp**. The **@+Tsp** means that the packet is sent to the **network** with a delay of **Tsp** time units. The transition also returns the packet to its incoming place, **DataToSend**.

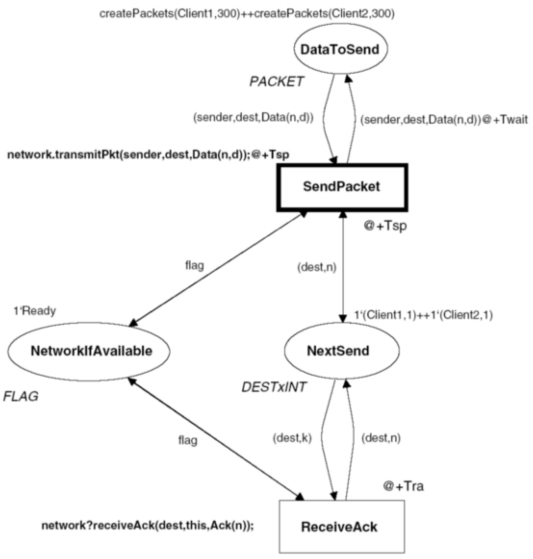
createPackets(Client1,300)++createPackets(Client2,300)
DataToSend
PACKET
(sender,dest,Data(n,d))
(sender,dest,Data(n,d))@+Twait
network.transmitPkt(sender,dest,Data(n,d));@+Tsp
SendPacket
@+Tsp
flag
(dest,n)
1'Ready
1'(Client1,1)++1'(Client2,1)
NetworkIfAvailable
NextSend
FLAG
DESTxINT
flag
(dest,k)
(dest,n)
@+Tra
network?receiveAck(dest,this,Ack(n));
ReceiveAck

Fig. 9. The Server Model

But the returned token has a timestamp **Twait** units greater than the current *model time*, as indicated by the TCP-net arc expression **(sender, dest, Data(n,d))@+Twait**. In practice, it means that the same packet is not resent before **Twait** time units (this is the time the server is configured to wait for an *ack* message from the client). The **NetworkIfAvailable** place holds a single token. This token is just a *timed* **flag**, which indicates whether the network interface is available for any operation.

The TCP-net inscription **@+Tsp** next to the **SendPacket** transition indicates that the **flag** token returns to its incoming place with a timestamp **Tsp** units of time greater than the model time, so the server is not able to perform any network operation (**SendPacket**,**ReceiveAck**) for **Tsp** time units.

It is important to observe that time was modeled at both the OO level (sending packets to the network) and the internal level (waiting to resend). It is also remarkable that different messages, possibly attached to the same transition, may have different time delays and these time delays may be the result of any valid expression. This allows the modeling of activities with delays that follow statistical distributions like Exponential, Normal, Poisson, etc.

Let us consider the **ReceiveAck** transition labeled with the RPOO input inscription **network?receiveAck(dest,this,Ack(n))**. The **this** in the inscription means that a Server object only takes messages whose destination is itself. The transition receives the ack message and *updates* the token coming from **NextSend** place so, the **SendPacket** transition will be able to send the next packet (the packet number **n**) in the sequence (for the client that sent the *ack* message). Also, the timestamp of the **flag** token from the **Network-IfAvailable** place is set to *model time*+**Tra**, indicating that this operation *uses* the server network interface for **Tra** time units.

The Client net shown in Figure 10 is modeled by two places and one single transition.

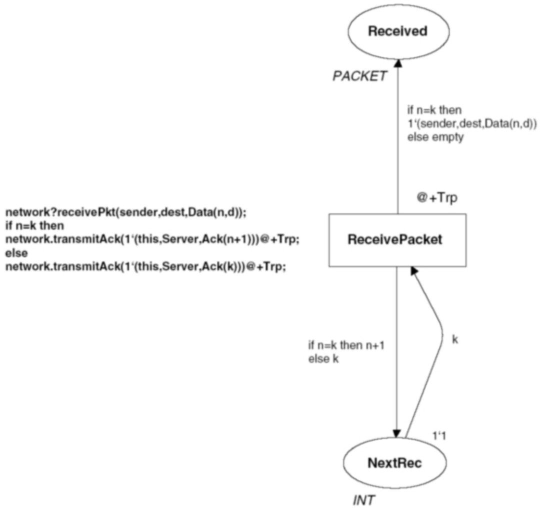
Received
PACKET
if n=k then
1'(sender,dest,Data(n,d))
else empty
@+Trp
network?receivePkt(sender,dest,Data(n,d));
if n=k then
network.transmitAck(1'(this,Server,Ack(n+1)))@+Trp;
else
network.transmitAck(1'(this,Server,Ack(k)))@+Trp;
ReceivePacket
k
if n=k then n+1
else k
1'1
NextRec
INT

Fig. 10. The Client Model

The **Received** place holds the packets delivered by the Network object (originally sent by a Server object). The **NextRec** place controls the sequence number of the packet that the client is waiting. The **ReceivePacket** transition receives a data packet from a object identified by **network** and sends an ack message (addressed to the sender of the data message) through the same **network** object, with a delay of **Trp** time units (see RPOOt inscription next to the transition). If the sequence number of the arriving packet (**n**) is equal to the expected one (**k**), an ack numbered with **n+1** is sent. So, the server can start to send the next packet on its sequence. Otherwise, an ack numbered with **k** is sent: this may happen because the object **network** eventually drops ack packets, causing the server to resend a data packet that arrived correctly at the client side.

The simplest model in this study is the Network one, shown in Figure 11. This class performs the task of dropping some packets or transmitting them with a random time delay. A Network object may also reorder packets that arrived at the same time. The function **Ok()** is used to promote packet loss. It randomly returns *true* or *false*.

The binding for **sender**, **dest** and **content** depends on the current con-

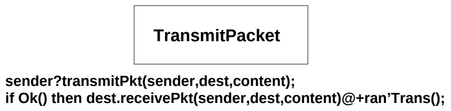
TransmitPacket
sender?transmitPkt(sender,dest,content);
if Ok() then dest.receivePkt(sender,dest,content)@+ran'Trans();

Fig. 11. The Network Model

figuration of the RPOOt object system. In practice, **sender** and **dest** can be bound to either the server or one of the clients. These objects are in the *structure* of the initial configuration of the object system. Configurations are depicted as directed graphs, where circles are objects, arcs represent object links and pending messages are drawn as rectangles. Figure 12 shows a possible initial configuration graph, where there are no pending messages and links from an object **network** to objects **server**, **client1** and **client2**.

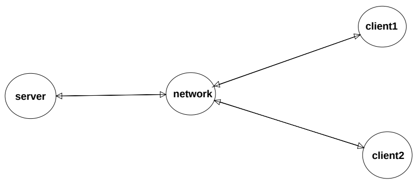

Fig. 12. A RPOO Configuration Graph

Note that the model of Network (Figure 11) can be easily reused to *transmit* packets to any other objects that implement the following interface: *void receivePkt(pkt:PACKET)*. If the language used to describe the datatype *PACKET* offers some degree of polymorphism, the net can be used in different models, for transmitting (and eventually dropping) a variety of concrete types of packets. Since the RPOOt models were implemented using the Design/CPN tool, the packets that may be sent, dropped and/or reordered by a Network object are restricted to the ones defined in the color set of the equivalent CPN model.

4 Validation and Timing Analysis

4.1 Validation

Both logical and timing analysis were conducted. The main objective of the protocol logical analysis was to remark that RPOOt models can be verified as RPOO models, since it is not necessary time information to check protocol logics. For instance, the correctness properties defined in Section 3 do not concern explicit, quantitative time constraints. These properties were translated into ML functions and formally verified using the Veritas RPOO model checker [14] — a tool that verifies properties over RPOO state spaces, providing proper counter examples whenever is the case.

Recall the main correctness property enumerated in Section 3:

- Any packet having a sequence number n always (**universally**) arrives at the client side **before** the packet with sequence number $n + 1$.

Since the Veritas checker implements property specification design patterns [7], it is not mandatory to know a lot about CTL (logic commonly used to express desirable properties of systems) to verify the models. For example: from the property listed above, it is necessary an atomic proposition (packet number n arrives) to *universally* occur *before* another one (packet number n+1 arrives). ML functions are written for each atomic proposition and is asked the checker to apply the property pattern *universally-before*. The functions for the atomic propositions take an RPOO occurrence graph node as parameter and return true or false. The checker also gives a truth-value after it evaluates the whole property. The protocol property previously defined went true.

It is also possible to define *explicitly timed* properties by means of CCTL formulas to be verified over RPOOt timed state spaces (CCTL is a *clocked* extension to CTL for time constrained property specification).

4.2 Timing Analysis

Roughly speaking, performance analysis of models can fall into one of two techniques: i) simulation models, where the measures concern a particular simulation, during which some data is collected and processed (or stored for further processing); ii) analytical models, which provide exact measures concerning not only a specific simulation but general execution scenarios.

There is a trade-off between these two perspectives. The latter seems to be more interesting at first sight. However, its application often requires the use of *restricted* mathematical models (like Markovian chains). As a consequence, the model may become a less realistic one because it is necessary to make simplifications and abstract away some characteristics of the modeled system in order to fit in some particular mathematical constraint. For these reasons, it becomes difficult to apply analytical models to complex real world systems. Simulation models, in turn, are more flexible in that they do not need to make restrictive assumptions about the modeled system, but they may be time consuming to simulate and it may be difficult to achieve accurate results [19]. RPOOt models are not analytical models. As stated previously, it is possible to analyze general executing scenarios by checking CCTL properties over timed state spaces, but the results are not as exact as those from analytical models.

Let us now consider timing analysis of RPOOt by means of *simulation*. The selected scenario to be simulated considers a server sending packets to several clients through the network. For each client, 500 packets were addressed.

The intention of this case study is not testing this protocol to use it in some real application. The simulation was performed in order to show that we can evaluate the performance of systems with models that implement the definitions of RPOOt. So, the time units in Figure 13 are just used as an example: they do not verify any particular situation.

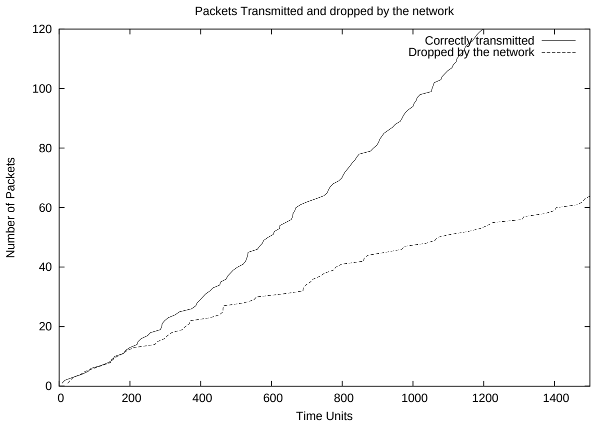

Fig. 13. Packets correctly transmitted and packets dropped

The data for the graphics was collected by means of simulations using Design/CPN and its *performance libraries*. This was done since there is no tool support for RPOO timed models direct simulation yet. The RPOOt models were translated into CPN-equivalent models. There is an algorithm [8] for the translation (from *RPOO* to CPN). The adaptation to RPOOt and future simulation/timing analysis was possible because RPOOt timing strategy can be easily integrated with the TCP-net models implemented by Design/CPN. Fortunately, TCP-nets were used to describe most of the classes. Also, all the clocks were synchronized: those from each class and the one from the object system were in fact the same clock.

The graphic in Figure 13 shows some performance measures for the protocol. One curve shows the number of packets that were correctly transmitted through the network, by time units. The other shows the number of dropped packets, also by time units.

Until time 200, the number of transmitted and dropped packets are almost the same, so the network correctly transmitted nearly 50 percent of the incoming packets. Some time units after that, the number of correctly transmitted packets starts to grow faster than the number of dropped ones, so the network performance increases. The network performance is shown in Figure 14. Nearly 70 percent of the incoming packets were correctly transmitted at time 1500. Figure 15 shows the network performance for the rest of the simulation.

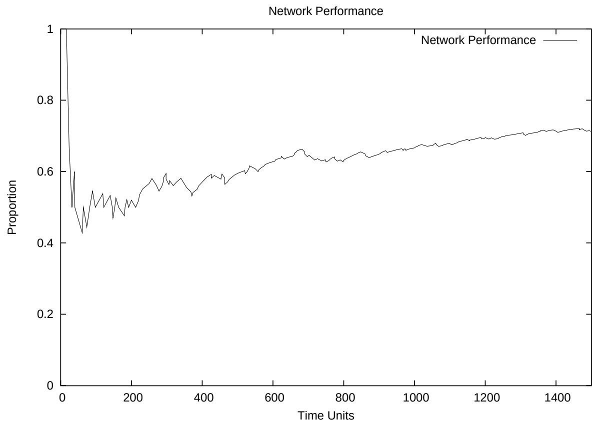

Fig. 14. Network performance

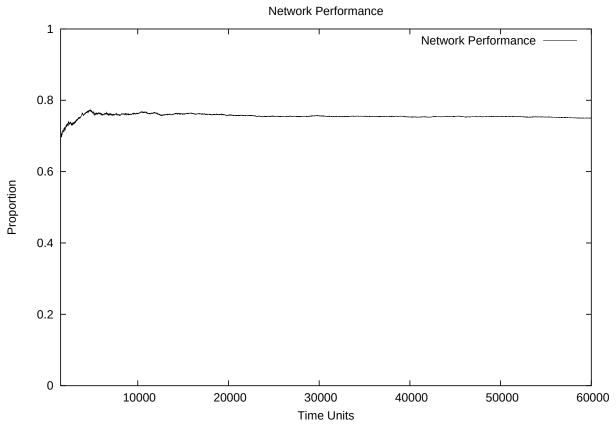

Fig. 15. Network performance

Now, different scenarios were considered, by changing the number of clients, in order to see how this affects the performance of the system. Considering scenarios going from 1 to 5 clients, it was observed the number of outgoing packets by time units for a fixed client (which we called *client 1*), in order to verify the effects of the addition of extra hosts in the data flow to the fixed client. The number of outgoing packets was slightly different, indicating that the system performs well with 5 clients in parallel. The results of these simulations are resumed in Figure 16.

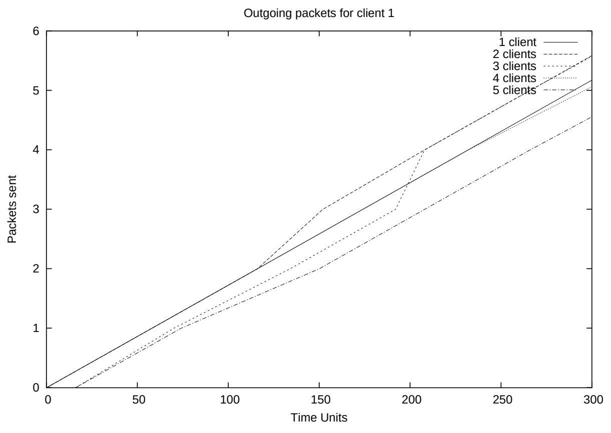

Fig. 16. Outgoing packets for a fixed client

5 The RPOO modeling language

An RPOO model consists of a set of classes and their corresponding Petri nets. Classes are described and can be related to each other in the conventional way—in fact, one may think of them as UML classes. The Petri nets describe the behavior of objects. Thus, a class and its corresponding net provide a formal model for a set of objects. In most OO languages, from a pragmatic point of view, an object represents a relevant entity from the world under study. In RPOO, as in other concurrent OO modeling languages, an object is an abstraction of *one or more* real entities. Therefore, a single object may represent a complete subsystem or component. This is the main reason for which objects are inherently concurrent—each object may encapsulate many threads of execution. Hence, RPOO provides for both intra- and inter-object concurrency.

The principle of encapsulation is strictly adhered to RPOO. Any interaction between objects must satisfy a declared interface. This means no direct relationship is allowed between any two nets of the model[5]. Interaction specification is addressed by means of an inscription language that allows to express inter-dependencies between objects *actions*. Action inscriptions are written in a syntax similar to other object-oriented ones. Possible actions allow objects to: send and receive messages, both in synchronous and asynchronous modes; instantiate and delete objects; and to create and eliminate links. From a for-

[5] In other approaches to object-oriented Petri nets, structural connection between class nets is used as a means to obtain larger models from smaller ones. However, this structural approach to composition breaks the encapsulation law and turns compositional formal analysis really hard to achieve, because the number of possible connections grows exponentially with the size of the net [8]

mal perspective, inscriptions allow the modeler to specify a partial relation over the possible behavior of the objects in a system.

5.1 RPOO Semantics

As with other OO languages, classes and relationships work as templates for instances of the model. An instance of an RPOO model is called a *configuration* and consists of a collection of linked objects—usually depicted as directed graphs. Intuitively, a configuration represents a given state of the modeled system. Formally, a configuration has four elements:

(i) a *structure*, consisting of objects, links and pending messages in the system in the given state;

(ii) a *type relation* that attributes a class to each object in the structure;

(iii) the *internal states function* that attributes a possible internal state to each object in the configuration;

(iv) and the *model* itself, consisting of the classes, their relationships and the nets;

For simplicity, we write only the structure, as in general the other parts can be easily determined from the context. In particular, it is important to note that, as nets are used to specify the behavior of objects, *reachable markings* of the nets can be used as internal states. The internal states function maps objects to markings of the correct type nets, as allowed by the type relation.

Structures can be expressed in an algebraic notation. The following grammar defines its abstract syntax:

$$E \ ::= \ 0 \ \mid \ x[x\ y\ \ldots\ z] \ \mid \ E + m_y^x \ \mid \ E + E$$

The first rule defines the empty structure, denoted by 0. It represents a configuration with no object and no pending messages. The second one defines the syntax for objects and their links. Observe that each object x in the system is represented as a $x[x\ldots]$ term in the structure. Within the square brackets, all object references stored in x are enumerated (in no special order). Observe that every object has a link to itself. However, for simplicity, self-links in expressions are omitted, thus $x[x\ y]$ is congruent to $x[y]$. We also define $x[x]$ as congruent to x, meaning that one can write x for objects that have no links to other objects. The third rule defines the syntax for messages. A message m_y^x has m as message contents and x and y as the sender and recipient objects, respectively. The fourth rule defines structure composition. The mentioned congruency relation is defined as the smallest relation for which the $+$ operator (defined over structures) is a commutative monoid with 0 as the neutral element.

An *effect relation* is defined over structures to cope with the effect of object actions. We write $(E, a) \rightarrow E'$ to express that the occurrence of action a transforms the structure E into structure E'. Six rules define the $\rightarrow$ relation, corresponding to the six possible actions. The axioms for asynchronous message sending and receiving are the following:

$$\begin{array}{ll} (1) & (E + x[y], x{:}y.m) \rightarrow E + x[y] + m_y^x \\ (2) & \quad (E + y + m_y^x, y{:}x?m) \rightarrow E + y, \end{array}$$

where E is any structure, $x{:}y.m$ denotes an action performed by object x that consists of sending an asynchronous message m to object y; and $y{:}x?m$ denotes an action performed by object y that consists of receiving message m sent by x. Other rules cope with the remaining possible actions and their composition.

A complete operational semantics can be obtained for RPOO by synchronizing object behavior (nets) and structure effects. Formally, we define a *direct reachability relation* over possible configurations, denoted by $\vdash$, and write $C \vdash_e C'$ to denote that configuration C' is directly reachable from configuration C if event e occurs—we usually omit e and write only $C \vdash C'$. A single rule defines the relation:

$$(3) \qquad \frac{(E, e) \rightarrow E' \qquad \sigma[T\rangle\sigma'}{(E, \sigma) \vdash_e (E', \sigma')}, \quad \text{where, } e = [\![I(T)]\!].$$

Above, E is a structure, σ is the internal state function, T is a set of transitions of class nets, and $I(T)$ is the set of action inscriptions of transitions in T. Thus, $[\![I(T)]\!]$ is the interpretation of the inscriptions as actions in the form used in the axioms above. The notation $\sigma[T\rangle\sigma'$ is a set extension for the classic notation for Petri nets events. It is a compact way to express that σ' is the set of internal states (markings) reached from σ if all transitions in T are fired. Recall that class nets are not structurally connected, thus all transitions are inherently concurrent.

The rule above states that a system in configuration (E, σ) can reach (E', σ') by means of an event e if it consists of the actions specified by the interaction inscriptions of some set of transitions T. For notation convenience, we also define a generalized reachability relation $\overset{*}{\vdash}$ as the reflexive and transitive closure of $\vdash$. As mentioned before, we usually omit internal states and refer to structures as configurations. Thus, we write $E \vdash_e E'$ and assume σ and σ' as appropriate.

6 A Timed Extension for RPOO Modeling Language

In the previous section, an overview of the RPOO formalism was presented. Here, the fundamental concepts and definitions of its timed extension, named RPOOt, are presented.

The definitions start by extending the formalism so that *pending* messages in RPOO *structures* are defined as timed multi-sets (in RPOO, they are *conventional* multi-sets). Next, rules for configuration evolution are adapted so that time parameters are properly considered.

Here is the definition for timed multi-sets:

Definition 6.1 A timed multi-set tm, over a non-empty set S, is a function $tm : S \times \mathbb{R} \rightarrow \mathbb{N}$ such that the sum:

$$tm(s) = \sum_{r \in \mathbb{R}} tm(s, r)$$

is finite for all $s \in S$. The non-negative integer tm(s) is the number of appearances of the element s in the timed multi-set tm. The list:

$tm[s] = [r_1, r_2, r_{tm(s)}]$ is defined to contain the time values $r \in \mathbb{R}$ for which $tm(s, r) \neq 0$.

We usually represent the timed multi-set tm by a formal sum:

$\sum_{s \in S} tm(s)`s@tm[s]$

In Section 5, it was presented how the *structure* of an object system in RPOO can be expressed as a formal sum. In what follows, the definition of structures is presented in details. Consider $\mathcal{O}$ a (potentially infinite) set of all objects and D a domain of data.

Definition 6.2 A structure of an object system is a tuple $\langle O, L, M \rangle$, where:

$O \subseteq \mathcal{O}$ is the set of *live* objects of the system;

L is a set of object links $\langle a1, a2 \rangle \in \mathcal{O} \times \mathcal{O}$;

M is a *timed* multi-set over $\mathcal{O} \times D \times \mathcal{O}$, the *pending* messages;

Timed configurations are defined as follows.

Definition 6.3 The tuple $\langle E, \sigma, \rightarrow, R, r \rangle$ is a timed configuration of an object system, where:

E is a structure (6.2) allowing the messages to be timed multi-sets over their domain rather than ordinary multi-sets;

σ is a function that maps each living object to its internal state;

$\rightarrow$ is a relation that represents the behavior of these objects;

R is a set of time values, a discrete subset of the reals closed under sum $(+)$;

$r \in R$ is the model time or configuration clock.

6.1 Effects of actions over a configuration

It is not part of this proposition the modeling of timed synchronous messages. This is due to the fact that the semantics of this kind of message does not involve the creation of new messages in the timed structure of the object system. This way, defining timed synchronous messages may imply in considering the internal state of the objects, and this can be done by choosing a timed Petri net model to describe the classes.

Moreover, timing asynchronous messages may be seen as a way of *synchronizing* the sending and receiving of messages, since the global configuration clock will not advance until all the messages that can be consumed at the current time are in fact consumed.

Here, the axioms for asynchronous sending and receiving of messages (see Section 5) are extended to cope with time.

$$(4) \qquad (E + x[y], A + x{:}y.m@t, r) \rightarrow (E + x[y] + m^x_y@t\ ,\ A,\ r)$$

$$(5) \quad (E + y + m^x_y@t, A + y{:}x?m@t, r) \rightarrow\ (E + y, A, r)\ \Leftrightarrow t \le r$$

$$(6) \qquad \begin{aligned} (E, A, r) \rightarrow (E, A, r') \Leftrightarrow \\ \nexists m^x_y@t \in E,\ y{:}x?m@t \in A.\ t \le r\ \mathit{and} \\ r' > r\ \mathit{and} \\ \forall m^x_y@t \in E, y{:}x?m@t \in A.\ r' \le t \end{aligned}$$

For simplicity, we omitted R and the σ function from the above configurations.

Assuming m as a timed message (resulting from the evaluation of some *term*), Axiom 4 assures that the messages m^x_y created in the structure by sending a message m must preserve the timestamp of m. Axiom 5 refers to the fact that only *ready* messages may actually be consumed. Ready messages are those which have timestamps (t) less than or equal to the *configuration time* r. Axiom 6 defines how the clock advances. It advances to the least value of time where a pending message can be consumed, provided that no pending messages can be consumed at the current clock value (r).

This way, sending a message with a timestamp n time units greater than the current configuration clock means that the message may *take* n time units to be sent. This happens because the message will be able to be actually delivered only after the clock equals its timestamp. However, if after n time units the internal state of the receiver object does not offer conditions for the message to be delivered, it may be pending for more the n time units (and it may never be delivered in some cases).

Afterall, an action of the type "$x : y.m@t$" means that the message will take *at least* t time units to be delivered. Note that the definitions of RPOOt are independent of the subjascent Petri Net model adopted. They are at an *OO level*. Considering these timed configurations and the effects of $\rightarrow$ over them, described by the axioms above, the definition of *direct reachability relation* remains the same that was described at the end of Section 5.

7 Conclusions

We presented the main characteristics of a definition for a formal timed OO Petri net based model, named RPOOt. The formalism was applied to the modeling of a simple network protocol, which was simulated and formally validated. Furthermore, some performance measures were taken as result from the simulation process.

The simplicity of the models showed that the formalism was suitable for the modeling of the time constrained stop-and-wait protocol. Though RPOOt concerns a number of formal expressions, the notation is relatively simple to the user. In particular, it can be specially suitable for the validation of complex systems because models are executable. Of course, this demands the development of tools that support the modeling, the simulation of models and the analysis of RPOOt models.

Our proposition differs from other timed OO models in different ways. RPOOt is formally defined and supports model checking, differently from the models presented by Selic[16] and Kruger [11]. It allows temporization *inside* objects and *also* at *OO level*, unlike the models from Bichler[2] and Shu[17]. We can also model time delays in a variety of statistical distributions (Normal, Exponential, Poisson, etc), which is not the case in all of these models.

RPOOt is independent of the Petri net model adopted to describe the classes. There is no restriction to the use of *untimed* nets to describe them. A closer look at the Network Petri net in Section 3 allows to see that it is not a TCP-net model, since it has no timed internal types (it has no places, in fact). This is a concrete example of a working model integrating a non-timed net/class with timed ones (Server and Client nets) via RPOOt formalism.

With our timed extension, RPOO language is provided with internal means for expressing time. Thus, time constrained functionalities can now be addressed and performance analysis by means of simulation can be achieved. However, the lack of proper tools to directly deal with RPOO/RPOOt models makes it harder to apply the formalism to more complex protocols like *Mo-*

bile IP. This way, experiments with more realistic protocols stand as future work. Currently, there are several developments on progress concerning tools to support RPOOt modeling, simulation and analysis.

References

[1] Azgomi, M. A. and A. Movaghar, *Towards an object-oriented extension for stochastic activity networks*, 10th Workshop on Algorithms and Tools for Petri Nets (2003), pp. 144–155.

[2] Bichler, L., A. Radermacher and A. Schurr, *Evaluating UML extensions for modeling real-time systems*, 2002.

[3] Boyer, M. and M. Diaz, *Non equivalence between time petri nets and time stream petri nets.*, in: *Proc. 8th Int. Workshop on Petri Net and Performance Models (PNPM'99), 8-10 October 1999, Zaragoza, Spain*, 1999, pp. 198–207.

[4] Canedo, E., J. A. Santos, J. C. de Figueiredo and D. D. S. Guerrero, *Experimenting a notation based on petri nets and object-oriented concepts.*, Proceedings of the I Brazilian Petri-Nets Meeting (2002).

[5] de A. Guerra, F. V., T. Silva, J. C. A. de Figueiredo and D. D. S. Guerrero, *Formal object-oriented modeling and validation of mobile IP protocol* (2003).

[6] de Figueiredo, J. C. A., "Redes de Petri com Temporizao Nebulosa," Ph.D. thesis, Curso de Pós-graduação em Engenharia Elétrica, Universidade Federal da Paraíba – Campus II, Campina Grande, Paraíba, Brasil (1994).

[7] Dwyer, M. B., G. S. Avrunin and J. C. Corbett, *Patterns in property specifications for finite-state verification*, Technical Report UM-CS-1998-035, University of Massachusetts Department of Information and Computer Science (1998).

[8] Guerrero, D. D. S., "Redes de Petri Orientadas a Objetos," Ph.D. thesis, Curso de Pós-graduação em Engenharia Elétrica, Universidade Federal da Paraíba – Campus II, Campina Grande, Paraíba, Brasil (2002).

[9] Jensen, K., "Coloured Petri Nets: Basic Concepts, Analysis, Methods and Practical Use. Volume 1," mtcs, Springer-Verlag, 1992.

[10] Jensen, K., "Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use. *Volume 2*," mtcs, Springer-Verlag, 1994.

[11] Kruger, I., W. Prenninger and R. Sandner, *Deriving architectural prototypes for a broadcasting system using UML-RT*, in: P. Kruchten, editor, *Proceedings 1st ICSE Workshop on Describing Software Architecture with UML*, 2001.

[12] Marotta, F., A. Merzenti and D. Mandrioli, *Modeling and analyzing real-time corba and supervision & control framework and applications*, in: *Proceedings of the The 21st International Conference on Distributed Computing Systems* (2001), p. 567.

[13] Perkins, C., *Rfc 3344:IP mobility support for IPv4* (2002), status: Proposed Standard. URL http://www.ietf.org/rfc/rfc3344.txt

[14] Rodrigues, C. L., J. C. A. de Figueiredo and D. D. S. Guerrero, *Verificacao de modelos em redes de petri orientadas a objetos*, in: *VII Workshop de Teses e Dissertacoes*, Manaus - Brasil, 2003.

[15] Santos, J. A., "Suporte a Análise e Verificação de Modelos RPOO," Master's thesis, Universidade Federal de Campina Grande, Campina Grande, Brasil (2003).

[16] Selic, B., *Tutorial: Real-time object-oriented modeling (room)*, in: *Proceedings of the 2nd IEEE Real-Time Technology and Applications Symposium*, 1996.

[17] Shu, G., C. Li, Q. Wang and M. Li, *Validating objected-oriented prototype of real-time systems with timed automata*, IEEE International Workshop on Rapid System Prototyping (2002).

[18] Taiani, F., M. Paludetto and J. Delatour, *Composing real-time objects: A case for petri nets and girard's linear logic*, in: *4th International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2001)* (2001), pp. 298–305.

[19] Wells, L., "Performance Analysis using Coloured Petri Nets," Ph.D. thesis, University of Aarhus, Aarhus, Denmark (2002).