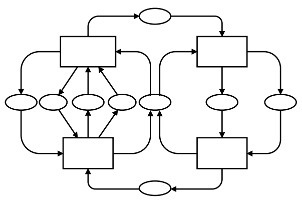
ISSN 0105-8517

# Petri Nets 2000



21st International Conference on
Application and Theory of Petri Nets

Aarhus, Denmark, June 26-30, 2000

Workshop Proceedings
**Software Engineering and Petri Nets**

Organised by

Mauro Pezzé
Sol M. Shatz

DAIMI PB – 548
June 2000

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF AARHUS
Ny Munkegade, Bldg. 540
DK-8000 Aarhus C, Denmark

# Preface

This booklet contains the proceedings of the Workshop on Software Engineering and Petri Nets (SEPN), held on June 26, 2000. This workshop was held in conjunction with the 21st International Conference on Application and Theory of Petri Nets (ICATPN-2000), organised by the CPN group of the Department of Computer Science, University of Aarhus, Denmark. The SEPN workshop papers are also available in electronic form via the web page: www.daimi.au.dk/pn2000/proceedings

The aim of the workshop was to bring together researchers and practitioners with interests in Petri nets and/or software engineering, with the goal of exploring more closely the potential impacts and pitfalls in applying net-based formalisms to software development problems.

All submitted papers were refereed and evaluated under the direction of a program committee with the following members:

Jonathan Billington, University of South Australia (Australia)
Ugo Buy, University of Illinois at Chicago (USA)
Robert France, Colorado State University (USA)
Dino Mandrioli, Politecnico di Milano (Italy)
Mauro Pezze, Politecnico di Milano (Italy)
Sol Shatz, University of Illinois at Chicago (USA)

The program for the workshop included ten selected papers and two invited talks. The invited speakers were: Professor Michal Young (Oregon State University, USA) and Professor Manfred Broy (Technishe Universitat Munchen, Germany).

Mauro Pezze and Sol Shatz
Co-organizers, SEPN-2000

# Table of Contents

# Performance Evaluation for the Design of Agent-based Systems: A Petri Net Approach[*]

José Merseguer, Javier Campos, and Eduardo Mena

Dpto. de Informática e Ingeniería de Sistemas, University of Zaragoza, Spain
{jmerse,jcampos,emena}@posta.unizar.es

**Abstract.** Software design and implementation using mobile agents are nowadays involved in a scepticism halo. There are researchers who question its utility because it could be a new technology that does provide new skills but it could introduce new problems. Security and performance are the most critical aspects for this new kind of software. In this paper we present a formal approach to analyse performance for this class of systems. Our approach is integrated in the early stages of the software development process. In this way, it is possible to predict the behaviour without the necessity to carry out the complete implementation phase. To show the approach, we model a software retrieval service system in a pragmatic way, later, the corresponding formal model is obtained and analysed in order to study performance.

**Keywords**: Software performance, Petri nets, UML, mobile agent

## 1 Introduction

In the last years, distributed software applications have increased their possibilities making use of Internet capabilities, positioning distributed software development as a very interesting approach. The client/server model has become the key paradigm to support distributed software development. It is widely recognised that there are four main technologies which advocate for client/server developments: relational database management systems (RDBMS), TP monitors, groupware and distributed objects. It is well accepted that distributed objects in conjunction with *mobile agents* [15, 11] technology are a very interesting approach to address certain kind of software domains like e-commerce, information retrieval and network management and administration.

Although there are researchers who question mobile software, it takes sense in distributed environments [9] because it is a technology with appropriate new skills for these kind of systems. But it could introduce new problems as the inappropriate use of the net resources. In this way time consuming could become a problem for users. So, we are concerned to develop new techniques and methods which minimize these problems. In this context, *software performance* [18] appears as a discipline inside software engineering to deal with model performance on software systems design. Like many people concerned about software

[*] This work has been developed within the project TAP98-0679 of the Spanish CICYT.

performance, we believe that the performance evaluation must be accomplished during the early stages of the software development process.

*Unified Modeling Language* (UML) [2] is widely accepted as a standard notation to model software systems. Unfortunately, UML lacks of the necessary expressiveness to accurately describe performance skills. There have been several approaches to solve this lack [19, 20, 16]. One of the goals of this paper is the study of the performance indices in mobile agent systems, thus, we propose a *UML with performance annotations* (pa-UML) to deal with performance skills on these kind of systems. Our approach to solve the problem is as follows: we model the problem domain using pa-UML, describing static and dynamic views when necessary. pa-UML models will give us the necessary background to obtain the corresponding formal model expressed as *Petri nets* [13]. From pa-UML, we derive a time interpretation of Petri nets leading to Generalized Stochastic Petri Nets (GSPN) [1]. Thus, we implicitly give a semantics for pa-UML in terms of Petri nets. Performance indices may be computed for GSPN by applying quantitative analysis techniques already developed in the literature.

The rest of the paper is organised as follows. In section 2, we describe a system, based on agents, which has been taken from [12]. In section 3, we give our proposal to annotate system performance aspects in UML (pa-UML) and we develop the pa-UML models for the system presented in section 2. Section 4 is dedicated to transform pa-UML diagrams into Petri nets in order to achieve the desired formal model. Finally, some performance results and conclusions are presented.

## 2 An example: the Software Retrieval Service in the ANTARCTICA system

In this section we briefly present ANTARCTICA[1]. The system has been taken from [12] and it will be used as an example along this paper to study performance on mobile agent systems.

The goal of the system is to provide mobile computer users with different services that enhance the capabilities of their computer. One of these services is the Software Retrieval Service, that allows users to select and download new software in an easy and efficient way. This service has been thought to work in a wireless network media and provides several interesting features:

- The system manages the knowledge needed to retrieve software without user intervention, using an ontology.
- The location and access method to remote software is transparent to users.
- There is a “catalog” browsing feature to help user in software selection.
- The system maintains up to date the information related to the available software.

In the following, we briefly describe the system paying attention in its components. There is a “majordomo” named *Alfred*, which is an agent specialised in

[1] Autonomous ageNT bAsed aRChitecture for cusTomized mobIle Computing Assistance.

user interaction. There is a *Software Manager* agent whose task is to create a catalog which will help the user to select the required software. Another agent, the *Browser* will help the user in selecting the software. Finally, a *Salesman* agent is in charge of performing any action previous to the installation of the selected software, like e-commerce.

The system was proposed in [12] using different technologies, namely CORBA [14], HTTP and mobile agents. Some performance tests were applied to different implementations, in order to select the best way of accessing remote software. Conclusions were the following:

- Time corresponding to CORBA and mobile implementations are almost identical for a wide range of files to be downloaded.
- Mobile agent approaches are fast enough to compete with client/server approach.

Although considering the importance and the relevance of the results of the work [12], we would like to stress the enormous cost of implementing different prototypes in order to evaluate the performance of the different alternatives. In the rest of this paper, we model the system in a pragmatic way using pa-UML, annotating consistently the system load (we have annotated the system load taking as a basis the experiments and experience of the authors of the cited paper). After that, we can interpret the pa-UML model in terms of Petri nets and derive the corresponding performance model which will be properly analysed. This analysis is used to evaluate the system.

## 3 Modelling the system using pa-UML

In the previous section, we have explained the general features of the target system. Now, we focus on modelling it using pa-UML notation. We have considered UML and not the notation of methodologies such as OMT [7], OOSE [10] or Fusion [6] because of its wider acceptance in the software engineering community.

The system description in UML accomplishes with static and dynamic views in order to give a complete description of the system. For the sake of simplicity and for the convenience of our problem, we only describe the dynamic view of the system.

Figure 1 shows the use cases needed to describe the dynamic behaviour of the system. We deal with three different use cases, “show services”, “software retrieval service” and “e-commerce”. Also, we can see the unique actor which interacts with the system, the “user”. The use cases are described in the following.

**Show services use case description.**

- Principal event flow: the use case goal is to show to the user the available services that the system offers. The Software Retrieval Service is one of those services and it is also described as a use case.

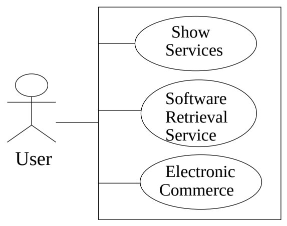


**Fig. 1.** Use Cases

**Software retrieval service use case description.**

- Principal event flow: the user requests the system for the desired software. The Browser gets a catalog and the majordomo, Alfred, shows it to the user, who selects the software s/he needs.
- Exceptional event flow: if the user is not satisfied with the catalog presented, s/he can ask for a refinement. This process could be repeated as many times as necessary until the user selects a concrete piece of software.

**Electronic commerce use case description.**

- Principal event flow: the goal is to provide the user an e-commerce activity and the download of the software selected.

Show services and e-commerce use cases are out of the scope of this article, thus, we concentrate on the Software Retrieval Service.

Pragmatic object-oriented methodologies such as [6, 7, 10] do not deal with performance skills. So, we can say that there is not an accepted *method* to model and study system performance in the object-oriented software development process. This lack implies that there is not a well-defined language or *notation* to annotate system load, system delays and routing rates. On the contrary, formal specification languages, such as LOTOS [17], or Petri nets [13], have considered and studied the problem in depth. Thus, there are several proposals where we can learn from.

As we remarked, it is our objective to propose a UML extension (pa-UML) to deal with performance on the software development process at the design stage. We consider that our proposal must accomplish with both, the method and the notation. First, the method will give us the process to model the system and the relevant parameters to be taken into account. We advocate for a pattern-oriented approximation. Lately, design patterns [8] have gained relevance in software development due to their simplicity and flexibility. But this will be subject of future research. Second, concerning the notation, it will be treated in this work.

In order to have a complete performance notation, the UML behavioural and structural models must be considered. Also, performance will play a prominent role in the implementation diagrams. In this paper, we are interested only in behavioural aspects, concretely in the sequence diagram and the state transition diagrams. Future works will deal with the rest of the UML diagrams to describe behaviour (use case diagrams, activity diagrams, collaboration diagrams), structural aspects, and implementation diagrams.

The UML notation to deal with time is based on the use of time restrictions. This restrictions are expressed as time functions on message names, e.g., {(messageOne.receiveTime - messageOne.sendTime) < 1 sec.}. We consider more realistic to annotate the message size. In this way, we could calculate performance for different net speeds.

## 3.1 Sequence diagrams

In order to understand the problem, it is interesting a more detailed description of the Software Retrieval Service use case. Thus, a *sequence diagram* [2] has been developed to treat accurately the mentioned use case, see figure 2.

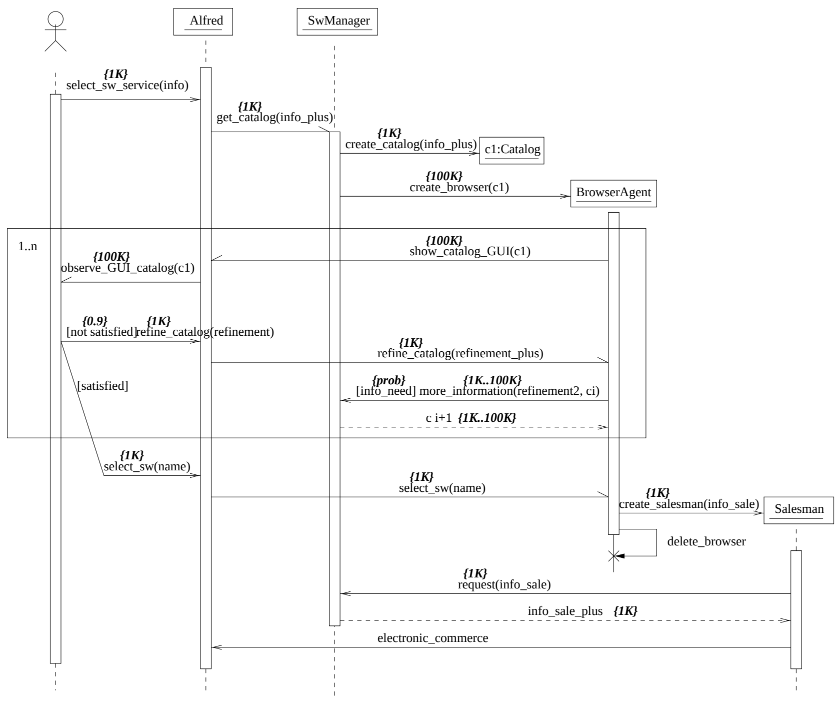


**Fig. 2.** Sequence diagram for the Software Retrieval Service use case

A sequence diagram represents messages sent among objects. Usually, a message is considered as no time consuming in the scope of the modelled system. But in a mobile agent system, we distinguish between messages sent by objects on the same computer and messages sent among objects on different computers, those which travel through the net. The first kind of messages will be considered as no-time consuming. The second kind will consume time as a function of the message size and the net performance (speed). Here an annotation, inside braces, will be made indicating the message size. For instance, in Figure

2, select_sw_service message is labelled with {1 Kbyte}, while show_catalog_GUI requires the movement of {100 Kbytes}. Also, it will be possible to annotate a range for the size in the UML common way, like in more_information message, where a {1K..100K} label appears.

In a sequence diagram, conditions represent the possibility that the message that they have associated with could be sent. An annotation, also inside braces, expressing the event probability success will be associated to each condition. A range is accepted too. See, for instance, the probability {0.9} associated in Figure 2 to the condition not_satisfied. Sometimes, it is possible that the probability is unknown when modelling. Also, it could be that the probability a message occurs is a parameter subject to study. In our example, the condition info_need associated to the more_information message is critical for the system, because it reveals how much intelligent the Browser is; so, we want to study it. In such situations, we will annotate an identifier, corresponding to the unknown probability.

## 3.2 State Transition diagrams

Sequence diagrams show how objects interact, but to take a complete view of the system dynamics, it is also interesting to understand the life of objects. In UML, the *state transition* diagram is the tool that describes this aspect of the system. For each class with relevant dynamic behaviour a state transition diagram must be specified.

In a state transition diagram two elements will be considered, the *activities* and the *guards*. Activities represent tasks performed by an object in a given state. Such activities consume computation time that must be measured and annotated. The annotation will be inside braces showing the time needed to perform it. If it is necessary, a minimum and a maximum values could be annotated. See, for example, bold labels between braces in Figures 4, 5, 6 and 7. Guards show conditions in a transition that must hold in order to fire the corresponding event. A probability must be associated to them. It will be annotated in the same way as guards were annotated in the sequence diagram, and the same considerations must be taken into account. See, for instance, label {0.9} joined to condition [not ^user.satisfied] in Figure 4.

Message size may be omitted since this information appears in the sequence diagram. In the example, we have duplicated this information to gain readability.

We now present the state transition diagrams for our system using the pa-UML notation.

**User state transition diagram.** In Figure 3, the behaviour of a user is represented. The user is in the wait state until s/he activates a select_sw_service event. This event sets the user in the waiting_for_catalog state. The observe_GUI_catalog event, sent by Alfred, allows the user to examine the catalog to look for the desired software, if it is in the catalog, the users selects the select_sw event, in other case s/he selects the refine_catalog event.

**Alfred state transition diagram.** The example supposes that Alfred is always present in the system, no creation event is relevant for our purposes. So the state transition diagram begins when a view_services event is sent

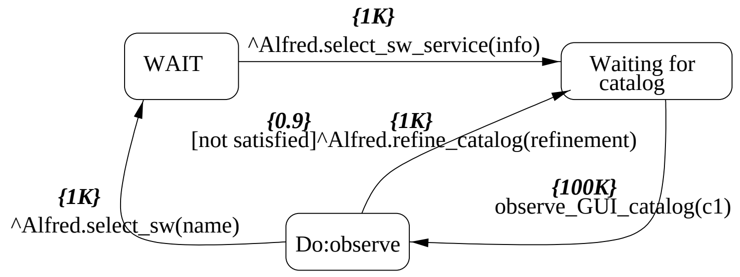


**Fig. 3.** State transition diagram for the user

to the user. Alfred's behaviour is typical for a server object behaviour. It waits for an event requesting a service (select_sw_service, show_catalog_GUI, refine_catalog or select_sw). For each of these requests it performs a concrete action, and when it is completed, a message is sent to the corresponding object in order to complete the task. After the message is sent, Alfred returns to its wait state to serve another request. Figure 4 shows Alfred's behaviour. The stereotyped transition ≪ *more_services* ≫ means that Alfred may attend other services that are not of interest here.

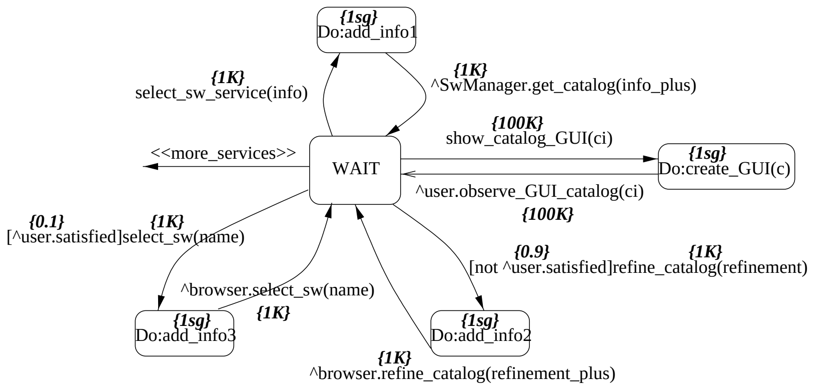


**Fig. 4.** State transition diagram for Alfred

**Software Manager state transition diagram.** Like Alfred, the Software Manager behaves as an server object. It is waiting for a request event (more_information, get_catalog, request) to enable the actions to accomplish the task. Figure 5 shows its state transition diagram; it is interesting to note the actions performed to respond the get_catalog request. First, an ontology is consulted and, after that, two different objects are created, those involved in task management.

**Browser state transition diagram.** The state transition diagram in Figure 6 describes the Browser's life. It is as follows: once the Browser is created it

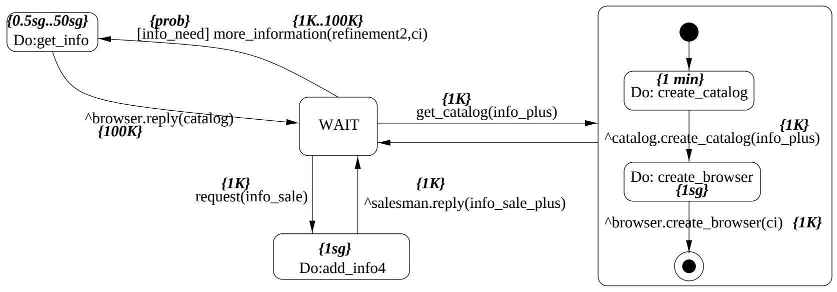


**Fig. 5.** State transition diagram for the Software Manager

must go to the MU_Place, where it invokes Alfred's shows_catalog_GUI method to visualize the previously obtained catalog. At this state it can attend two different events, refine_catalog or select_sw. If the first event occurs there are two different possibilities: first, if the Browser has the necessary knowledge to solve the task, a refinement action is directly performed; second, if it currently has not this background, the Browser must obtain information from the Software Manager, by sending a more_information request or by travelling to the software place. If the select_sw event occurs, the Browser must create a Salesman instance and die.

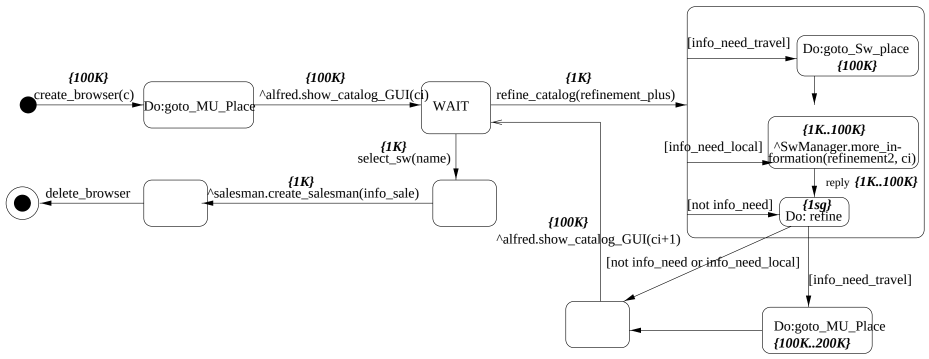


**Fig. 6.** State transition diagram for the Browser

**Salesman State Transition Diagram.** The Salesman's goal is to give e-commerce services, as we can see in Figure 7. After its creation it asks the Software Manager for sale information. With this information the e-commerce can start. This is a complex task that must be described with its own use case and sequence diagram which is out of the scope of this paper.

The pa-UML models that we have developed are expressive enough to accomplish with different implementations. A necessary condition to design methods

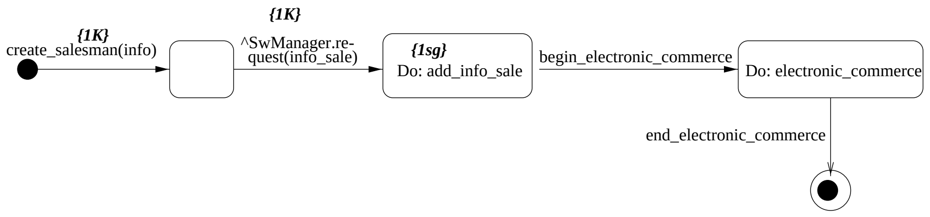


**Fig. 7.** State transition diagram for the Salesman

is their independence of final implementation decisions. In that way, we can use these models to develop applications based on CORBA, mobile agents, etc. But this gap between design and implementation could be undesirable in certain cases. For example, in the system that we are treating we are not sure how many majordomos should attend requests, how many concurrent users can use the system, etc. However, a formal modelling with Petri nets solves these questions satisfactorily.

The design proposed in [12] deals with one user and one majordomo. Petri nets allow to represent cases such as:

1. One user and one majordomo (the proposed system).
2. Several users served by one majordomo.
3. Many users served by many majordomos, once per request.

Thus, increasing the modelling effort, it could be possible to avoid the necessity of implementing the system for predicting performance figures.

## 4 Modelling with Petri nets

At this point, we have modelled the system with pa-UML notation, taking into account the load in the sequence diagram and the state transition diagrams. So, a pragmatic approach of the system has been obtained. But this representation is not precise enough to express our needs. Remember that we want to predict the system behaviour in different ways. First, we want to study how the system works with only one user served by one majordomo. On the other hand, it is also of our interest to know the system behaviour when several users are served by only one majordomo, or by several majordomos.

In order to obtain answers to our questions, we need to apply performance analytic techniques to the developed pa-UML diagrams. But there is a lack in this field because no performance model exist for UML, so the pragmatic model is not expressive enough. Also, we need to express system concurrency, but UML models concurrency in a very poor way. Thus, it is required a formal model of the system with concurrency capabilities.

To solve these lacks, we have chosen Petri nets as formal model, because it has the remarked capabilities and also there are well-known analytic techniques to study system performance in stochastic Petri net models. Thus, we propose some *transformation rules* to obtain Petri nets from pa-UML diagrams.

In the following, we model with Petri nets the first two proposed systems, the third one will be developed in a future work. For the first system, one user and one majordomo, GSPN have the expressive power to accomplish the task. To study the second system, several users served by one majordomo, stochastic well-formed coloured Petri nets [3] are of interest. Once the systems are modelled, we use analytic techniques implemented in GreatSPN [4] tool to obtain the target performance requirements.

### 4.1 Petri net model for a system with one majordomo and one user

First, we are going to obtain a Petri net for each system class, the *component nets*. Obviously, every annotated state transition diagram will give us the guide, and the following general transformation rules will be applied:

**Rule 1.** *Two different kinds of transitions can be identified in a state transition diagram. Transitions which do not spend net resources and transitions which do. The first kind will be translated into "immediate" transitions (that fire in zero time) in the Petri net. The second kind will be "timed" transitions in the Petri net. The mean of the exponentially distributed random variable for the transition firing time will be calculated as a constant function of the message size and net speed. More elaborated proposals like those given in [5] could be taken into account, but we have considered more important to gain simplicity.*

**Rule 2.** *Actions inside a state of the state transition diagram are considered as time consuming, so in the Petri net model they will be consider as timed transitions. The time will be calculated from the CPU and disk operations needed to perform the action.*

**Rule 3.** *Guards in the state transition diagram will become immediate transitions with the associated corresponding probabilities for the resolution of conflicts.*

**Rule 4.** *States in the state transition diagram will be places in the Petri net. But there will be not the unique places in the net, because additional places will be needed as an input to conflicting immediate transitions (obtained by applying Rule 3).*

Figures 8, 9, 10, 11 and 12 represent the nets needed to model our *system components* taking into account the previous transformation rules. According to GSPN notation [1], immediate transitions (firing in zero time) are drawn as bars (filled), while timed transitions are depicted as boxes (unfilled). Timed transitions are annotated with firing rates, while immediate transitions are annotated with probabilities for conflict resolution.

The sequence diagram will be the guide to obtain a *complete* Petri net for the system using the previous component nets. We must consider that UML distinguishes, in a concurrent system, two different kind of messages in a sequence diagram:

– those represented by a full arrowhead (*wait semantics*), and

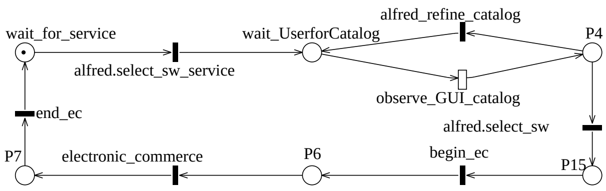


**Fig. 8.** User Petri net component

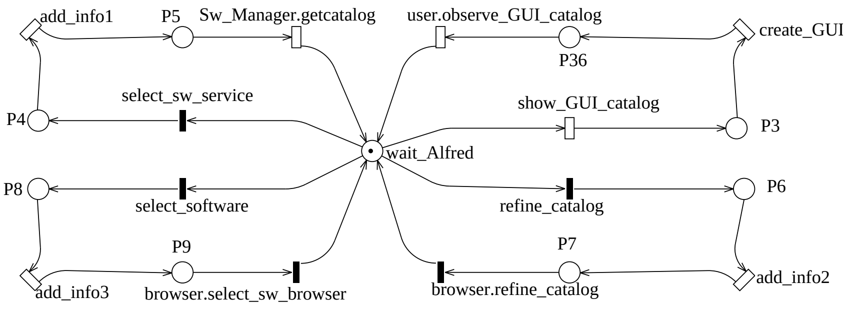


**Fig. 9.** Alfred Petri net component

– those represented by a half arrowhead (*no-wait semantics*).

The following transformation rules will be used to obtain the net system. But first, it must be taken into account that, for every message in the sequence diagram, there are two transitions with the same name in two different component nets, the net representing the sender and the net representing the receiver.

**Rule 5.** *If the message has wait semantics, only one transition will appear in the complete net system; this transition will support the incoming and outcoming arcs from both net components.*

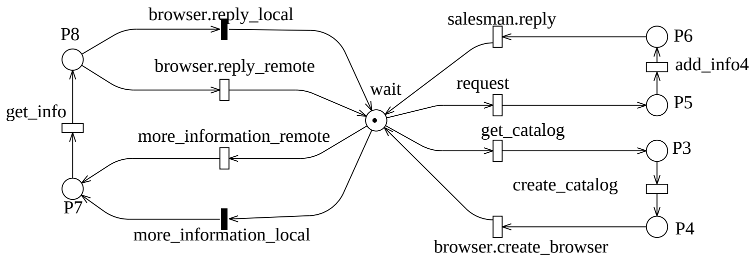


**Fig. 10.** Software Manager Petri net component

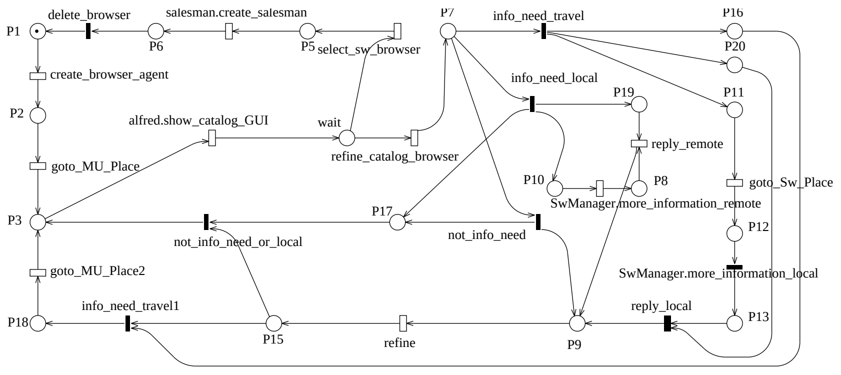


**Fig. 11.** Browser Petri net component

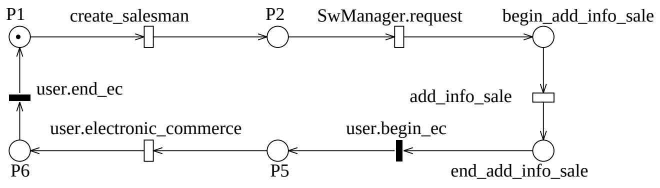


**Fig. 12.** Salesman Petri net component

**Rule 6.** *If the message has no-wait semantics, the two transitions will appear in the net system and also an extra place will be added modelling the communication buffer. This place will receive an arc from the sender transition and will add an arc to the receiver transition.*

The net system for the example is shown in Figure 13. In order to understand how to apply the previous rules, we are going to explain how to obtain the observe_GUI_catalog transition in the net system (Figure 13) from the observe_GUI_catalog message sent by Alfred to the user in the sequence diagram. We can observe in Alfred's net (Figure 9) and in the user's net (Figure 8) the presence of that transition. So, in the net system the transition appears with the union of the incoming and outcoming arcs of the components, synchronising in this way both objects.

Finally, we remark with an example that the concurrency expressed in UML has been achieved in the net system by synchronising component nets. When create_salesman transition fires one token is placed in P20 and one token is placed in P31, allowing a concurrent execution of the request and delete_browser transitions.

## 4.2 Petri net model for a system with one majordomo and several users

In order to model with Petri nets the situation of several users being served by one majordomo, we need to include several tokens in some places like, for

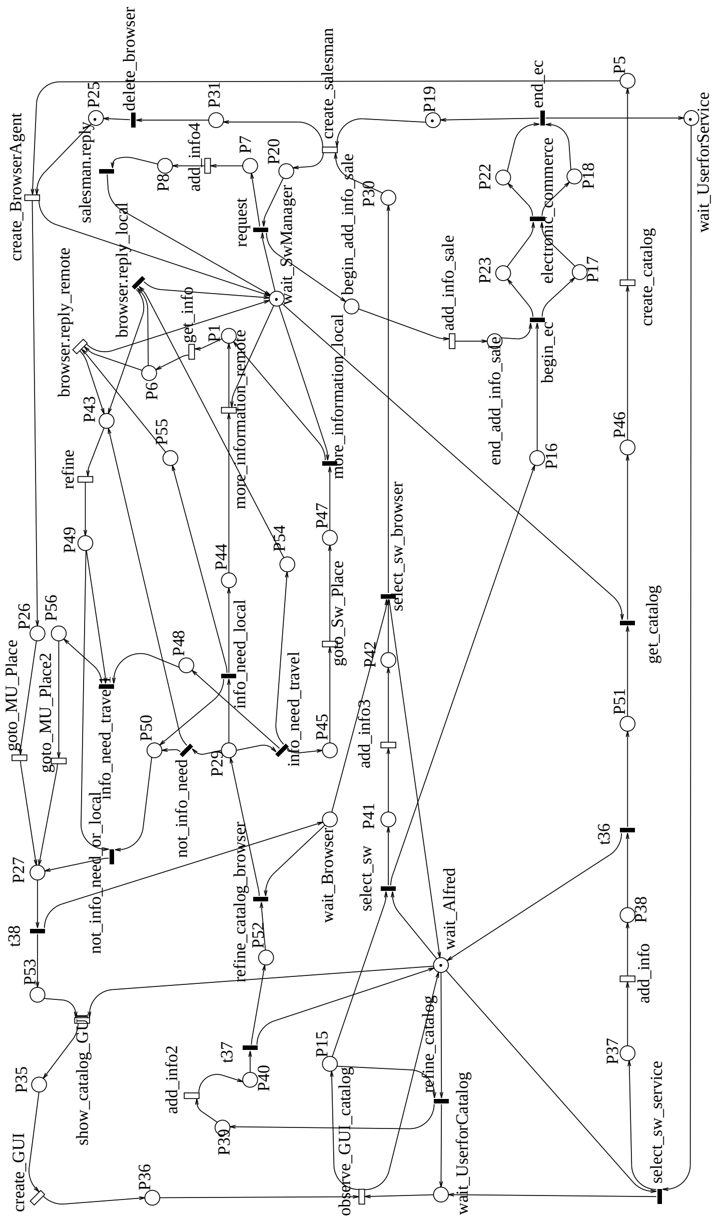


**Fig. 13.** The Petri net for the whole system

instance, wait_UserforService. Since the system must distinguish between different tokens (they represent different requests), we add a colour domain for the requests, thus leading to stochastic well-formed coloured Petri nets [3].

Our objective now is to reach the stochastic well-formed coloured nets for the components and for the system. Let us begin with the component nets. As in the previous system, component nets will be obtained from the annotated state transition diagrams. We begin the translation task (from pragmatic model to formal model) using the rules stated in the previous section. The Petri nets for Alfred and the Software Manager will be the same because only one instance of each is present in the system. On the contrary, the system will have as many instances of users, browsers and salesmen as required, suppose five for the example.

Now, pay attention on Figure 14, which represents the well-formed coloured Petri net for the user. The R colour means that the system deals with one to five requests and the initial marking m1 in place wait_for_service denotes that all class instances will be used. Moreover, all the places in the net have colour R and the arcs are labeled with the identity function (<x>), in this way only one request could be fired once a time.

Figure 15 shows the well-formed coloured Petri net for the Browser. It has been obtained applying the transformation rules to the Browser‘s STD. Initial marking m1 in place P1 shows that a maximum of five browsers could be created, one for each users request. Salesman well-formed coloured Petri net (see Figure 16) has been designed in the same way.

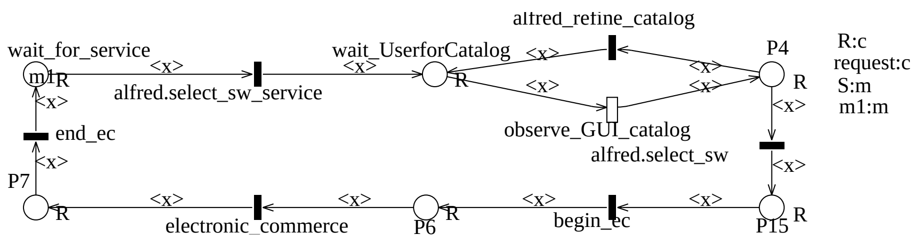


**Fig. 14.** User coloured Petri net component

Now, we are going to focus on the complete well-formed coloured net for the system, see Figure 17. The transformation rules given in the previous section will give us the guide to construct it. In addition, the following transformation rules will be applied concerning the colours:
**Rule 7.** *All colours and markings defined in the component nets will be inherited by the net system.*

**Rule 8.** *The places with colour and/or markings in the components nets will appear in the net system in the same way.*

**Rule 9.** *The arcs labelled in the component nets will appear in the net system in the same way.*

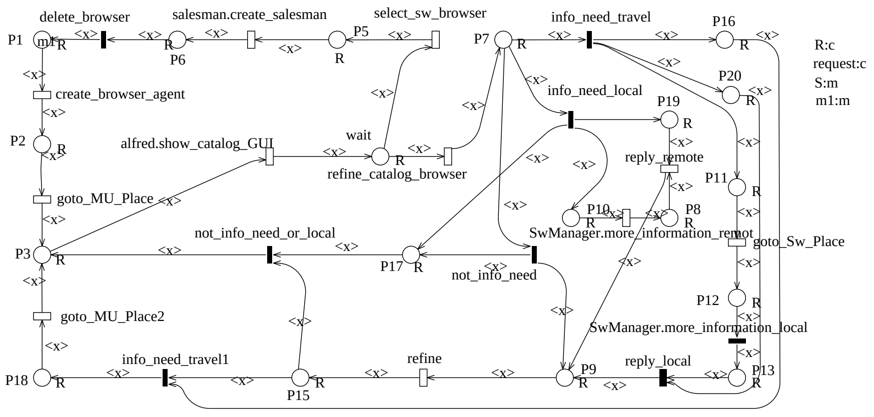


**Fig. 15.** Browser coloured Petri net component

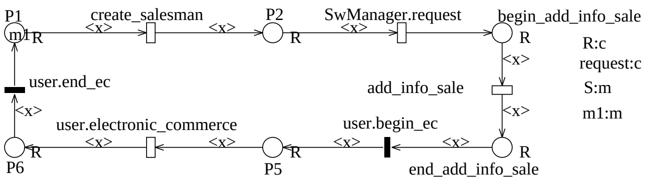


**Fig. 16.** Salesman coloured Petri net component

**Rule 10.** *Conflicting arcs are those that appear labelled in a component net but not in the component net which it is synchronised. When conflicting arcs appear, the net system must have the two arcs labelled, preserving in this way the richest semantic.*
As an example of Rule 9 see outcoming arcs for the synchronised transitions select_sw and alfred.select_sw in Figures 9 and 14 respectively.

We remark that the complete well-formed coloured net for the system describes concurrency at the same level as the complete net for the system given in the previous section. Moreover, it introduces a new level of concurrency. The use of coloured tokens models concurrent user requests of a complete service, as it can be seen in the select_sw_service transition, that can fire several tokens from place wait_UserforService representing several user requests.

## 5 Performance results

The results presented in this section have been obtained from the complete nets that model the examples; the complete net that models the case in which the system is used by one user, who is attended by only one majordomo and the complete net that models the case in which the system is used by several users, which are attended by only one majordomo.

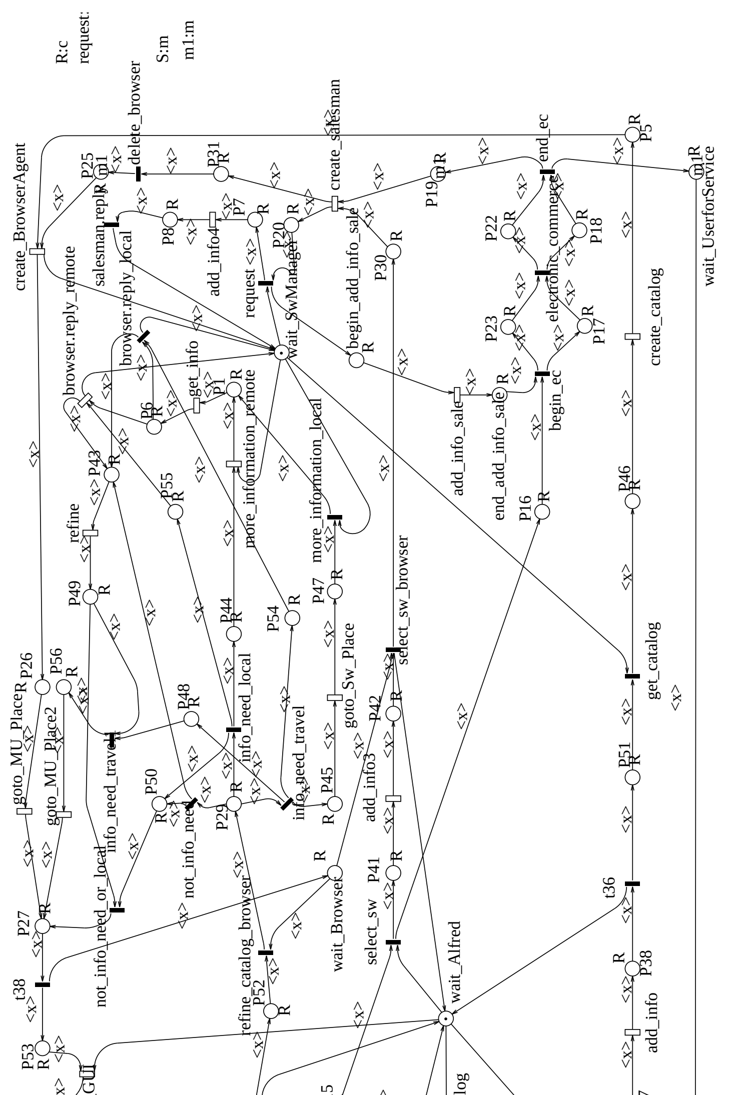


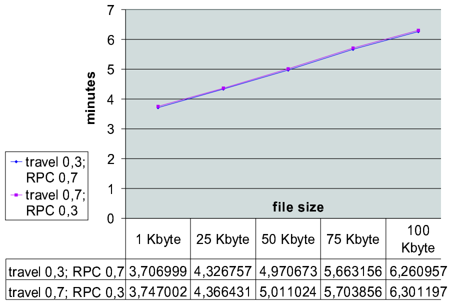
**Fig. 17.** The coloured Petri net for the whole system

It is of our interest to study the system *response time* in the presence of a user request. To obtain the response time, first the throughput of the select_sw_service transition, in the net system, will be calculated by computing the steady state distribution of the isomorphic *Continuous Time Markov Chain* (CTMC) with *GreatSPN* [4]; finally, the inverse of the previous result gives the system response time. *We want to know which are the bottlenecks of the system and identify their importance.* There are two possible parts which can decrease system performance. First, the trips of the Browser from the "user place" to the "software place" (and way back) in order to obtain new catalogs. Second, the user requests for catalog refinements, because s/he is not satisfied with it.

In order to study the two possible bottlenecks, we have developed a test taking into account the following possibilities:

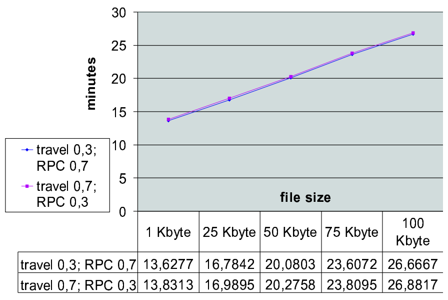
1. When *the Browser needs a new catalog* (under request of the user) there are several possibilities:
   - The Browser has enough information to accomplish the task or it needs to ask for the information. It is measured by the not_info_need transition. We have considered an "intelligent Browser" which does not need information the 70% of the times that the user asks for a refinement.
   - When the Browser needs information to perform the task, it may request it by a *remote procedure call* (RPC) (represented in the net system by the info_need_local transition) or it may travel through the net to the Software_place (represented in the net system by the info_need_travel transition) to get the information and then travel back to the MU_Place. In this case, we have considered two scenarios. First, a probability equal to 0.3 to perform a RPC, so a probability equal to 0.7 to travel through the net. Second, the opposite situation, a probability equal to 0.7 to perform a RPC, therefore a probability equal to 0.3 to travel through the net.
2. To test the *user refinement request*, we have considered two different possibilities. An "expert user" requesting a mean of 10 refinements, and a "naive user" requesting a mean of 50 refinements.
3. *The size of the catalog* obtained by the Browser can also decrease the system performance. We have used five different sizes for the catalog: 1 Kbyte, 25 Kbytes, 50 Kbytes, 75 Kbytes and 100 Kbytes.
4. *The speed of the net* is very important to identify bottlenecks. We have considered two cases: a net with a speed of 100 Kbytes/sec. ("fast" connection speed) and a net with a speed of 10 Kbytes/sec. ("slow" connection speed).

Figure 18(a) shows system response time (in minutes), for the net in Figure 13, supposing "fast" connection speed, "expert user" and an "intelligent" Browser. One of the lines represents a probability equal to 0.7 to travel and 0.3 to perform a RPC, the other line represents the opposite situation. We can observe that there are small differences between the RPC and travel strategies. Such a difference is due to the round trip of the agent. As the agent size does not change, this difference is not relevant for the global system performance. Thus, we show that the use of mobile agents for this task does not decrease the performance.



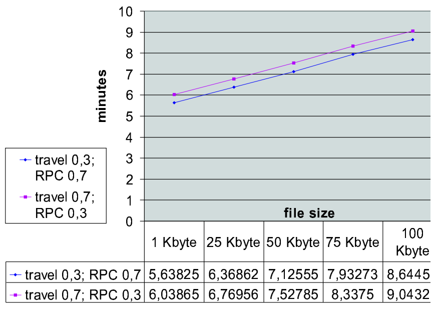
| | 1 Kbyte | 25 Kbyte | 50 Kbyte | 75 Kbyte | 100 Kbyte |
|---|---|---|---|---|---|
| travel 0,3; RPC 0,7 | 3,706999 | 4,326757 | 4,970673 | 5,663156 | 6,260957 |
| travel 0,7; RPC 0,3 | 3,747002 | 4,366431 | 5,011024 | 5,703856 | 6,301197 |

(a)



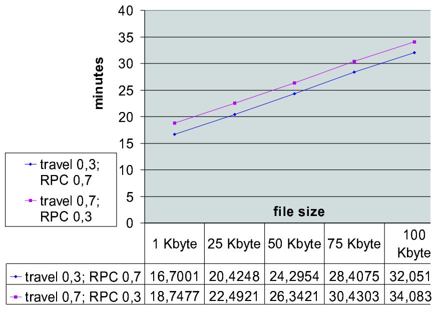
| | 1 Kbyte | 25 Kbyte | 50 Kbyte | 75 Kbyte | 100 Kbyte |
|---|---|---|---|---|---|
| travel 0,3; RPC 0,7 | 13,6277 | 16,7842 | 20,0803 | 23,6072 | 26,6667 |
| travel 0,7; RPC 0,3 | 13,8313 | 16,9895 | 20,2758 | 23,8095 | 26,8817 |

(b)



| | 1 Kbyte | 25 Kbyte | 50 Kbyte | 75 Kbyte | 100 Kbyte |
|---|---|---|---|---|---|
| travel 0,3; RPC 0,7 | 5,63825 | 6,36862 | 7,12555 | 7,93273 | 8,6445 |
| travel 0,7; RPC 0,3 | 6,03865 | 6,76956 | 7,52785 | 8,3375 | 9,0432 |

(c)



| | 1 Kbyte | 25 Kbyte | 50 Kbyte | 75 Kbyte | 100 Kbyte |
|---|---|---|---|---|---|
| travel 0,3; RPC 0,7 | 16,7001 | 20,4248 | 24,2954 | 28,4075 | 32,051 |
| travel 0,7; RPC 0,3 | 18,7477 | 22,4921 | 26,3421 | 30,4303 | 34,083 |

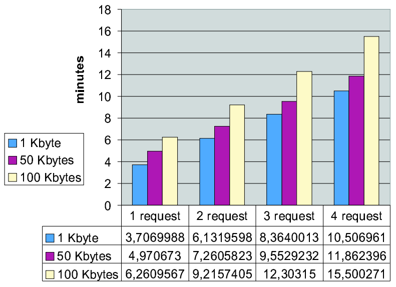
(d)

**Fig. 18.** Response time for a different scenarios with an "intelligent Browser". (a) and (b) represent a "fast" connection speed, (c) and (d) a "slow" connection speed; (a) and (c) an "expert user" and (b) and (d) a "naive user".

Figure 18(b) shows system response time (in minutes), supposing "fast connection", "intelligent" Browser, "naive user". The lines have identical meaning than in Figure 18(a). The two solutions still remain identical.

Someone could suspect that there exist small differences because of the net speed. So, we have decreased the net speed to 10 Kbytes/sec., (Figures 18(c) and 18(d)). It can be seen how the differences still remain non significant.

Finally, Figure 19 represents a test for an "intelligent Browser", an "expert" user, a probability for RPC equal to 0.7 and equal to 0.3 to travel. Now, we have tested the system for a different number of requests ranging from 1 to 4, thus the coloured model in Figure 17 has been used. Observe that when the number of requests is increased, the response time for each request increases, i.e., tasks cannot execute completely in parallel. Alfred and the Software Manager are not duplicated with simultaneous requests. Thus, they are the bottleneck for the designed system with respect to the number of concurrent requests of the service. Therefore, the next step in the performance analysis of the model would be to consider several majordomos (we do not include here due to space limitations).



| | 1 request | 2 request | 3 request | 4 request |
|---|---|---|---|---|
| 🟦 1 Kbyte | 3,7069988 | 6,1319598 | 8,3640013 | 10,506961 |
| 🟪 50 Kbytes | 4,970673 | 7,2605823 | 9,5529232 | 11,862396 |
| 🟨 100 Kbytes | 6,2609567 | 9,2157405 | 12,30315 | 15,500271 |

**Fig. 19.** Response time for an "intelligent Browser", an "expert user", a "fast" connection and also different number of request.

## 6 Conclusions and further work

The main goal of this paper was to present an approximation to evaluate performance in design mobile agent software. We have used as test a system designed for providing mobile computer users with a software retrieval service. We summarise the contributions in the following items:

– A model to evaluate software performance has been integrated in the software life cycle. It has been done in the early stages of the modelling process. Thus, when performance or functional requirements change, it will be easy and less expensive to assume them. Moreover, the approach will permit to obtain the performance figures in an automatic way: Starting from the pa-UML models, the component Petri nets are systematically achieved, and from these the net system, finally the net system allows performance evaluation.
– In order to apply any technique to analyse rigorously system performance, the use of a formal model is crucial. So, we have used Petri nets to design software, avoiding the UML ambiguity.
– Concurrency is ambiguously expressed in UML, but when the translation to Petri nets is performed, a concurrent well-defined model is gained, so different kinds of concurrent systems can be analysed.
– The modelled example presents a complex system which is expensive to implement. Our approach offers an analytic way of evaluating such kind of systems without having to implement several prototypes. The results coincide with those obtained by the ANTARCTICA designers. Their results were obtained with implemented prototypes.

Concerning future work, we are interested in the following objectives:

– Software design is a complex task. So, we advocate for the reuse of the knowledge acquired in the application domain. In this way, patterns will be introduced to design software using agents. Each design pattern will deal with its own performance skills. So, we will have a pattern design library with the proper use of the performance parameters.
– As we have said, UML semantics is not defined formally, so our approach brings a formal semantics based on Petri nets to model the system. In this article, we have proposed rules to obtain the Petri nets. We will work in this line to get a formal translation from the pa-UML notation to Petri nets semantics.

## References

[1] M. Ajmone Marsan, G. Balbo, and G. Conte, *A class of generalized stochastic Petri nets for the performance evaluation of multiprocessor systems*, ACM Transactions on Computer Systems **2** (1984), no. 2, 93–122.

[2] G. Booch, I. Jacobson, and J. Rumbaugh, *OMG Unified Modeling Language specification*, June 1999, version 1.3.

[3] G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad, *Stochastic well-formed coloured nets for symmetric modelling applications*, IEEE Transactions on Computers **42** (1993), no. 11, 1343–1360.

[4] G. Chiola, G. Franceschinis, R. Gaeta, and M. Ribaudo, *GreatSPN 1.7: GRaphical Editor and Analyzer for Timed and Stochastic Petri Nets*, Performance Evaluation **24** (1995), 47–68.

[5] J. Dilley, R. Friedrich, T. Jin, and J. Rolia, *Web server performance measurement and modeling techniques*, Performance Evaluation (1998), no. 33, 5–26.

[6] D. Coleman et Al., *Object oriented development. the Fusion method*, Object Oriented, Prentice Hall, 1994.

[7] J. Rumbaugh et Al., *Object oriented modeling and design*, Prentice-Hall, 1991.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of reusable object-oriented software*, Addison-Wesley, 1995.

[9] C. Harrison, D. Chess, and A. Kershenbaum, *Mobile agents: are they a good idea?*, Mobile Object Systems: Towards the Programmable Internet, 1997, pp. 46–48.

[10] I. Jacobson, M. Christenson, P. Jhonsson, and G. Overgaard, *Object-oriented software engineering: A use case driven approach*, Addison-Wesley, 1992.

[11] E. Kovacs, K. Röhrle, and M. Reich, *Mobile agents OnTheMove -integrating an agent system into the mobile middleware*, Acts Mobile Summit (Rhodos, Grece), June 1998.

[12] E. Mena, A. Illarramendi, and A. Goñi, *Customizable software retrieval facility for mobile computers using agents*, Proceedings of the 7th International Conference on Parallel and Distributed Systems (ICPADS'2000), Workshop International Flexible Networking and Cooperative Distributed Agents (FNCDA'2000) (Iwate (Japan)), IEEE Computer Society, July 2000.

[13] T. Murata, *Petri nets: Properties, analysis, and applications*, Proceedings of the IEEE **77** (1989), no. 4, 541–580.

[14] Object Management Group, *The common object request broker: Architecture and specification*, June 1999, Revision 2.3.

[15] E. Pitoura and G. Samaras, *Data management for mobile computing*, Kluwer Academic Publishers, 1998.

[16] R. Pooley and P. King, *The unified modeling language and performance engineering*, IEE Proceedings Software, IEE, March 1999.

[17] N. Rico and G.V. Bochman, *Performance description and analysis for distributed systems using a variant of LOTOS*, 10th International IFIP Symposium on Protocol Specification, Testing an Validation, July 1990.

[18] C. U. Smith, *Performance engineering of software systems*, The Sei Series in Software Engineering, Addisson–Wesley, 1990.

[19] G. Waters, P. Linington, D. Akehurst, and A. Symes, *Communications software performance prediction*, 13th UK Workshop on Performance Engineering of Computers and Telecommunication Systems (Ilkley), Demetres Kouvatsos Ed., July 1997, pp. 38/1–38/9.

[20] M. Woodside, C. Hrischuck, B. Selic, and S. Bayarov, *A wide band approach to integrating performance prediction into a software design environment*, Proceedings of the 1st International Workshop on Software Performance (WOSP'98), 1998.

# Testing Petri Nets for Mobile Robots Using Gröbner Bases

Angie Chandler[1], Anne Heyworth[2], Lynne Blair[1], Derek Seward[3].
[1] Lancaster University, Department of Computing
[2] University of Wales, Bangor, Department of Mathematics
[3] Lancaster University, Department of Engineering

## Abstract

As autonomous mobile robots grow increasingly complex, the need for a method of modeling and testing their control systems becomes greater. This paper discusses the use of Petri nets as a means of modeling and testing the control of a mobile robot, concentrating specifically on the reachability testing of the Petri net model through the use of Gröbner bases.

The designing and testing of the Petri net models for the mobile robot is done initially in component form, providing a model which is then automatically converted into a Gröbner basis to provide a simple means of reachability testing. Once the testing process is complete, the Petri net modules, which represent each of the components of the mobile robot are connected to form a single Petri net. This Petri net is then used for the generation of control code for the robot.

In this paper, the process of testing the modules created to represent the components of the autonomous mobile robot is shown through a case study, Star Track (a tracked autonomous mobile robot). Details of both ordinary and colored Petri nets representing certain components of Star Track are discussed, with both the Petri net model and the equivalent Gröbner basis described.

## 1 Introduction

The dynamic and asynchronous structure of the Petri net is ideally suited to the modeling of an autonomous mobile robot, provided the model can be thoroughly tested prior to code generation or execution on board the robot. To this end, the reachability test, as one of the most basic means for checking the accuracy of the model compared to its expected execution, provides a great deal of reassurance to the designer of the software, which in turn allows the designer to create more complex systems reliably.

The need for mathematical analysis of the Petri net models created for the mobile robot, provided an ideal opportunity for collaboration between mathematics and engineering departments. As a result of this co-operation, an approach to Petri net analysis formed, based on the relationship between Petri nets and Gröbner bases. The application of Gröbner bases has been successfully used in fields such as operational research and statistics, but is as yet less common in engineering.

The subject of this paper is the application of the Gröbner basis to the testing of reachability in a Petri net model, specifically to a Petri net model of a mobile robot. This application is implemented as an automatic testing facility within a Petri net toolkit, TRAMP (Toolkit for Rapid Autonomous Mobile robot Prototyping) intended to model mobile robots and other mechatronic systems from the stages of conceptual design to a final executable program. These Petri net models are initially formed as individual modules, each related to a component of the system, in order to allow easier testing and analysis prior to creation of the final, global, Petri net model [Chandler 99a].

In section 2 of this paper, some previous Petri net applications will be discussed, providing a background to the choice of Petri nets as a model for the autonomous mobile robot. Section 3 will detail the Petri net toolkit, TRAMP, before the Gröbner bases used as a testing method are studied in further detail (section 4). A case study showing the use of the method for the mobile robot, Star Track, will be discussed in section 5, followed by conclusions and future work.

## 2 Choice of Petri Nets

Petri nets are generic enough to provide the capacity for application to a wide variety of applications, although due to their asynchronous nature they are more commonly used for distributed systems [Buchholz 92] and other similar processes. However, their uses in distributed systems by no means exclude applications to the field of robotics. In fact, robots can themselves form part of a distributed system, as can be seen through the example of an orange-picking robot with point-to-point communications [Cavalieri 97]. Other areas of robotics can also find use for Petri net modelling as a method of eliminating deadlock and other temporal inconsistencies [Simon 98] [Caloini 98], although these properties require testing through timed Petri nets, an extension which has yet to be made to the analysis system used here. Alternatively, Petri nets can be used to allow co-operation between multiple robots [Suh 96], or between a human and a robot [Mascaro 98]. The range of applications for Petri nets is enormously diverse, and limited only by the range of tools available to implement these possibilities.

Our use of Petri nets as a modeling tool in the field of mobile robotics, was initially inspired by their ability to represent both the data flowing in the system and the state of the system, simultaneously. This initial interest was then furthered by the ease with which the model could be translated into executable code, as required by the TRAMP toolkit discussed in the section 3, below, without the need to alter any of the components modeled. These factors, combined with the mathematical background which supported the testing of any models used, and examples of previous applications to the field led to the eventual use of Petri nets within the TRAMP toolkit.

## 3 TRAMP

The TRAMP toolkit provides a simple means of modeling, testing and generating code for a mobile robot. This is initially done in the form of modules, or objects based on the separate components of the mobile robot, and divided into five categories in an overall object diagram in order to allow the toolkit user to connect the objects as desired. These five categories, sensors, filters, navigation, low level control, and actuators are also used to provide certain attributes to each object which may only be relevant to that category.

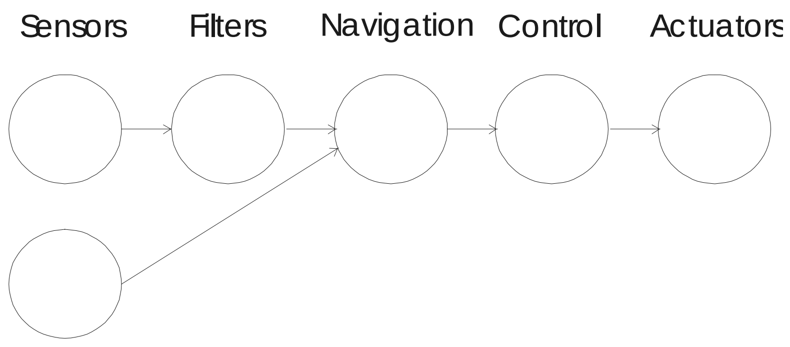


**Figure 1 Object Diagram Layout**

As the arcs in Figure 1 suggest, the flow of information in the system leads from sensors to actuators via various processing alternatives. Once data has been read in from a sensor, such as a compass, the data may then be filtered to remove any noise from the readings, before the navigation uses the data to make a decision on the next move of the robot. With the commands to be issued to the actuators decided, the navigation module will then pass the information either directly to the actuator (for example a motor) or via a low level controller, which will translate the information into a form readable by the actuator and ensure that it behaves exactly as it should.

Once the modules are defined and linked in the object diagram, as shown in Figure 1, the user may then access the Petri net modules of each of the separate components. These components remain completely unconnected whilst they are tested, which may include testing on board the robot as a separate module, after which the modules may be linked according to a precise protocol. This is discussed below.

## Linking

Once all testing of the Petri net is complete, the user may then link the individual components according to the connections defined in the object diagram, and a specific hierarchy. This hierarchy makes the navigation module the highest level element, and works outwards in the object diagram making the sensors and actuators the lowest. The navigation module (or modules) is designed so that whilst it can be tested in simulation as it stands, several of its transitions actually represent groups of transitions for use when the Petri net is finally connected. As the Petri nets are linked, the navigation module (Figure 2) fully expands its complex transitions.

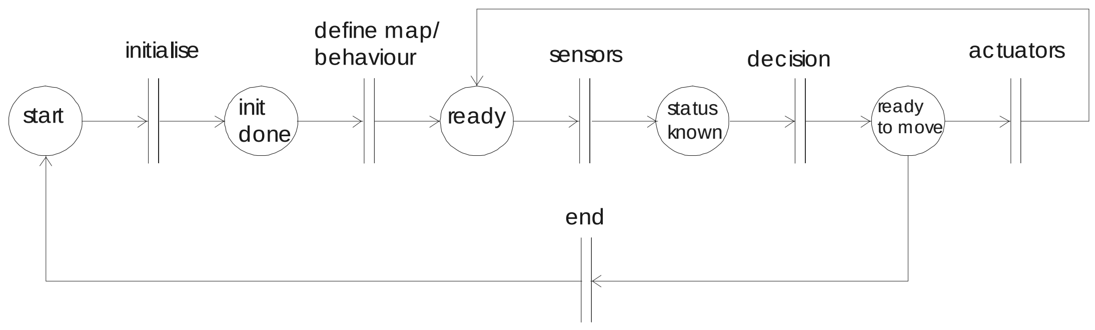


**Figure 2 Navigation Module**

The transitions “initialize”, “end”, “sensors” and “actuators” are all substitution transitions [Jensen 96], each expanding into several transitions, providing connections to the other modules. The remaining “define” and “decision” modules represent the actual method of navigation required of the robot, and can easily be exchanged for a number of standard defaults, or left for the user to fully implement.

## Initialization and End Expansion

The “initialize” and “end” transitions connect directly to every other module in the system, ensuring that every initialize routine is called before the main program starts, and that the program shuts down correctly when it finishes. Each initialize place shown here will be connected to a transition within the relevant module which is defined as an “initialize” transition and marked for connection outside the module in a method similar to that used in [Caloini 98]. The expansion of these is shown below, with an expanded view of the low level section of the motors Petri net.

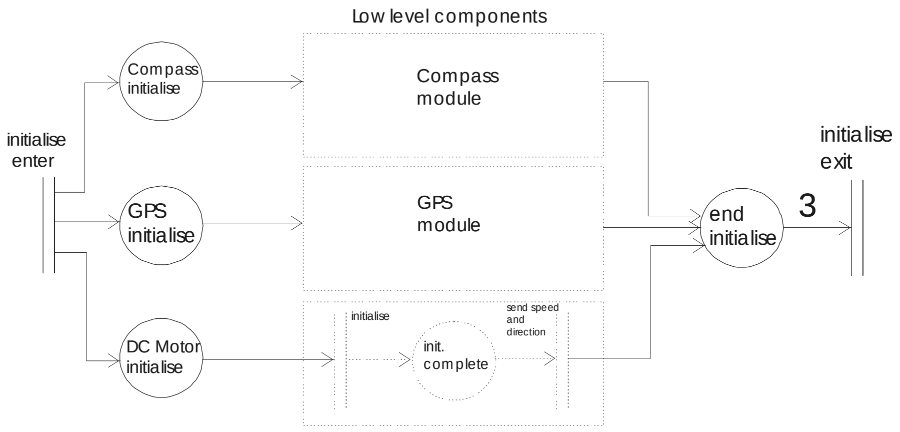


**Figure 3 Initialize Transition Expansion**

Figure 3 shows the expansion of the initialise transition to connect to two sensors (the compass and the GPS – Global Positioning System) and one actuator (the DC motor). Here, there were no filters or controllers in the system.

## Sensor and Actuator Expansion

Similarly, the "sensors" and "actuators" transitions can be expanded. However, here the links created in the object diagram come into play, as the only connections made are those which are directly connected to the navigation module. For the expansion of a "sensors" transition, this includes any filters which are connected to the navigation module, but not any sensors connected only to the filter as they must be connected through a similar process in the filter module.

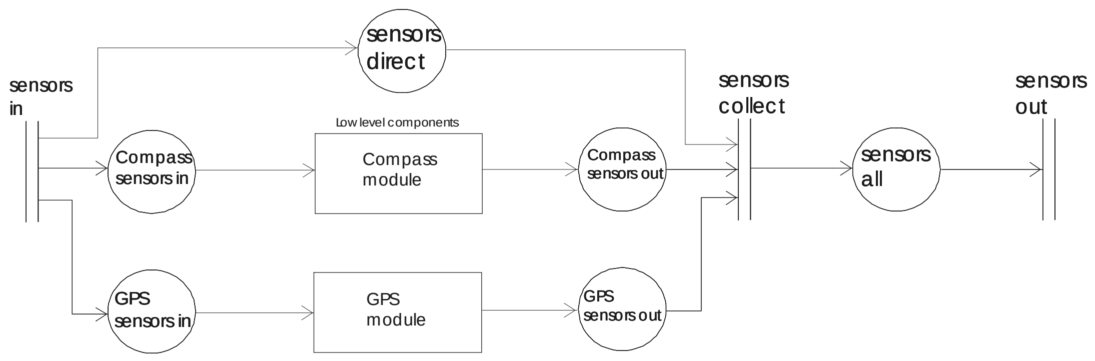


**Figure 4 Sensor Transition Expansion**

The configuration for sensor transition expansion (Figure 4) is slightly different to allow for the possibility that there many be no sensors or filters connected, but the transition must still operate. The places which link to the lower level modules behave as they do in the "initialize" transition expansion, connecting in place of the test driver initially provided with each module, to transitions which are defined from the start as an externally connecting. In this case, the connecting transitions of the compass can be seen in Figure 7, transitions "request data" and "send data".

There are also special standardized tokens, which allow this operation to be performed more smoothly. These tokens contain all possible elements of any expected sensor readings or instructions to actuators respectively. These readings or instructions can then be easily converted to or from the tokens created for specific sensor and actuator modules. Here, there were two sensors directly connected to the navigation module and no filters.

## Control from Navigation

The method of expansion described in the previous sub-sections is designed specifically to allow the navigation module to maintain control over the system as a whole. The bipartite nature of the Petri net gives the two types of nodes, places and transitions, specific meanings that must be taken into account within the model. Clearly, the transitions perform the actions, whereas the places merely maintain state, but there are further implications which can be put into use here. The nature of the places is such that they have authority over transitions. Transitions cannot fire without, in a sense, instructions from a place as each input place must contain a token in order to enable the transition. It is analogous to the handing out of instructions by a superior, and prior to the receipt of permission the task may not be performed.

Whenever a link is formed between two modules, the module which is further up the hierarchy contains the place which is linked, whilst the lower level module contains the transition. The lower level module knows only that an instruction has been received, whilst the higher level module continues to be aware of its state, despite the departure of the active tokens into a separate module.

This predictable method of linking also ensures that the reachability test results performed whilst the modules were still separated will still be accurate once the modules are reconnected, as the individual modules remain essentially separate whilst the pre-tested navigation module and connectors form links to them.

## 4 Testing through Gröbner Bases

Gröbner basis theory is a branch of computer algebra which provides methods for solving problems of equivalence in various types of algebraic structure. In the commutative case, computational Gröbner basis methods have been successfully applied in theorem proving, robotics, image processing, coding theory and signal processing, amongst others [Buchberger 98] [Holt 96]. All major computer algebra packages now include implementations of these procedures and there are also pocket calculator implementations. A formal definition of the Gröbner basis is included as an appendix to this paper. We also refer the reader to [Fröberg 97] for further details.

In this paper, the application of Gröbner basis procedures to the problem of reachability testing is discussed for reversible Petri nets and demonstrated with a practical case study.

The generation of the Gröbner basis of a set of polynomials, as is used for the Petri net analysis in later sections, is done with the use of Buchberger's algorithm [Buchberger 98]. The algorithm calculates a Gröbner basis for a set of polynomials by repeatedly testing and appending it with further polynomials until the appended set satisfies the properties of a Gröbner basis with respect to a chosen well-ordering of the variables. This method is more formally defined in the appendix.

Once created, the Gröbner basis may be used to find any reachable marking, provided the initial marking is a home marking, or alternatively determine whether the Petri net is reversible based on results of reachability testing (see section 5).

## 5 Case Study – Star Track

**Figure 5 Star Track**

The Petri nets discussed here are based on real life models, created for use on board the mobile robot, Star Track (Figure 5), intended to perform navigation with the use of satellite GPS [Yavuz 99]. This robot's major components consisted of a compass, a GPS receiver, a PC 104 computer, and four DC motors. These four components formed the main objects within the object diagram, two of which are considered in the following examples. It should be noted that the Petri nets shown represent the software interface to the hardware components named, not the hardware components themselves, as the intention of TRAMP is the generation of control software for use with specific hardware.

## Motors

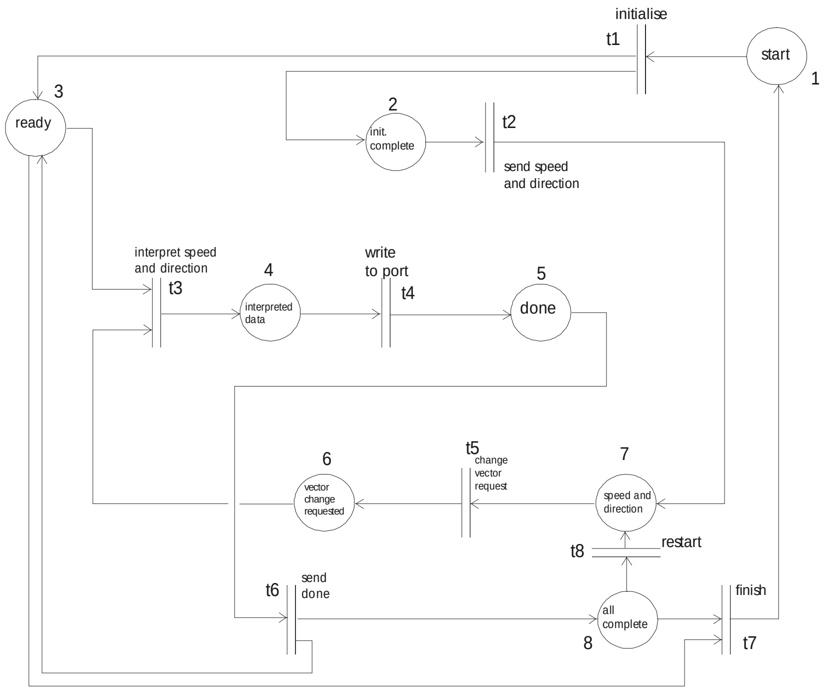


**Figure 6 Motors Petri net**

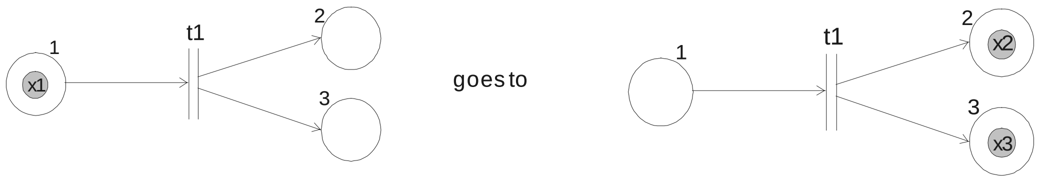
As can be seen in the Petri net shown in Figure 6, once the motors have been initialised ($t_1$) the user may input the required speed and direction ($t_2$) for each motor. The speed and direction information is then interpreted ($t_3$) and written to the relevant port ($t_4$), provided the system is "ready" (place 3) which, combined with the user input token, will enable transition $t_3$.

The Gröbner basis for this Petri net is generated from the polynomials of the transitions listed below, where x represents a token in a given place.

For example, a token at place 1 allows the firing of transition $t_1$ and results in a token in places 2 and 3. This can be represented by the polynomial:

$$\text{pol}(t1) = x_1 - x_2x_3$$

And seen in the diagram below:



Polynomials for the other transitions can be similarly generated:

$$\text{pol}(t2) = x_2 – x_7$$
$$\text{pol}(t3) = x_3x_6 – x_4$$
$$\text{pol}(t4) = x_4 – x_5$$
$$\text{pol}(t5) = x_7 – x_6$$
$$\text{pol}(t6) = x_5 – x_3x_8$$
$$\text{pol}(t7) = x_3x_8 – x_1$$
$$\text{pol}(t8) = x_8 – x_7$$

These polynomials are then used to form the Gröbner basis:

$$\{x_4 – x_1, x_5 – x_1, x_6 – x_2, x_7 – x_2, x_8 – x_2, x_2x_3 – x_1\}$$

This Gröbner basis can be calculated automatically, either through TRAMP or through a standard package such as Maple.

This gives a catalogue of markings (reachable places) from an initial marking $x_1$ (i.e., starting with a token in the place "start") to be:

$$\{x_1, x_4, x_5, x_2x_3, x_3, x_6, x_3x_7, x_3x_8\}$$

These reachable markings are found based on their equivalence to the defined initial marking modulo the transitions, which can be determined algorithmically, using polynomial reduction with respect to the Gröbner basis (see appendix).

As the Gröbner basis is only useable when the Petri net is reversible, an undesirable, or unexpected state within this list would indicate either that the Petri net was not reversible, or that there was an error in the Petri net itself, allowing the unexpected state. Once the possibility of either an undesirable reachable place (or alternatively a desirable place which wasn't reached) or a non-reversible Petri net is eliminated, the chance of a serious error occurring on board the mobile robot during execution is greatly reduced.

Should an undesirable state be reachable from the initial marking, it must first be decided whether the error is in the reversibility of the Petri net or in the reachability. This is best done by checking for errors in very simple, and obvious, reachability calculations. If the tester claims that a clearly unreachable state is reachable then it is likely that the Petri net is in fact not reversible, and that is where the error lies. Should the Petri net appear to be reversible, the user can seek further assistance by using the "step through" method, which gives a visual representation of the movement of the tokens through the Petri net, and allows the user to see the error as it occurs.

## Compass

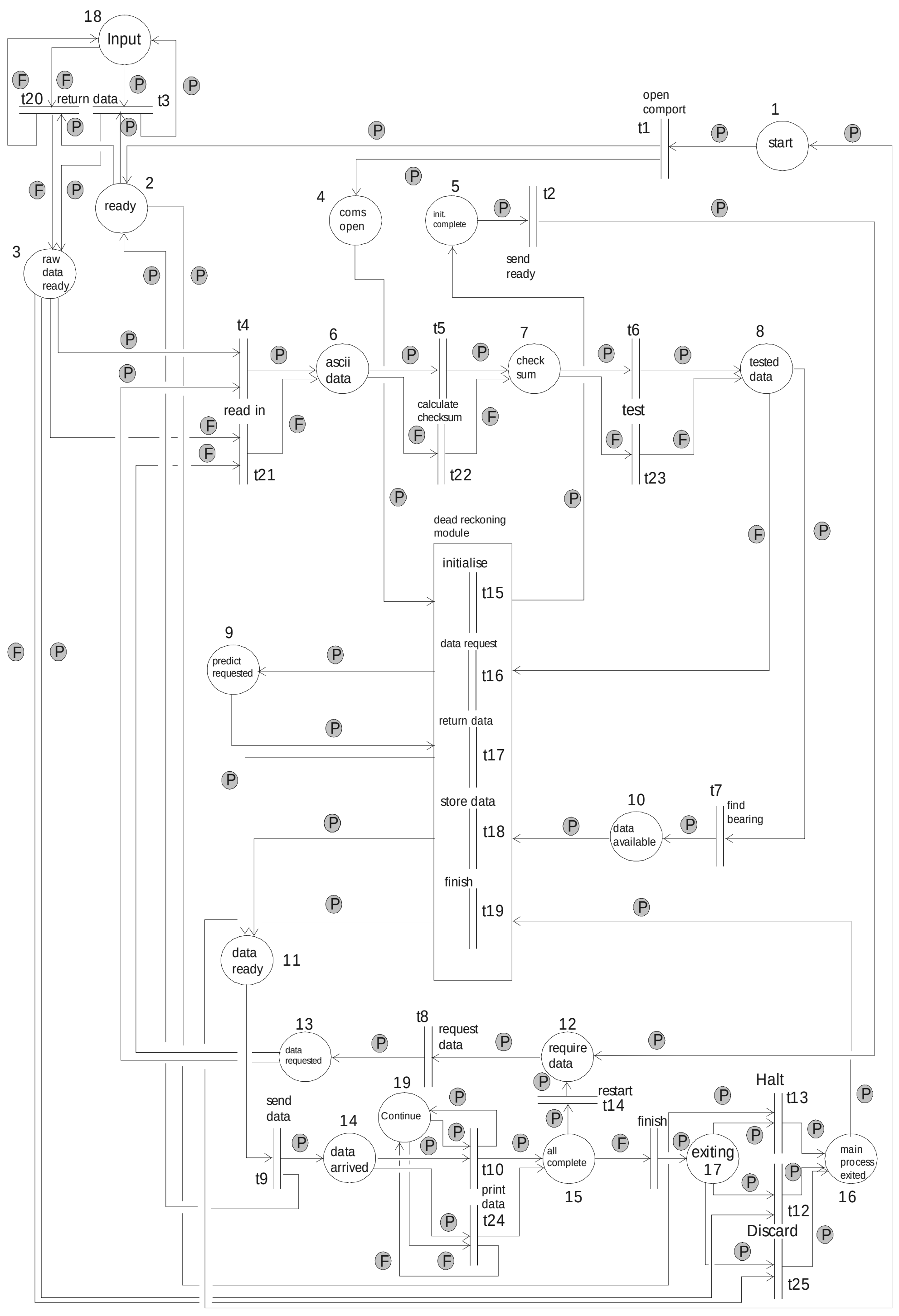


**Figure 7 Compass Petri Net**

Before analyzing the Petri net shown in Figure 7, it is necessary to consider a further extension to the Petri net theory described in section 2. The compass Petri net is a *coloured Petri net* with a finite set of colors.

The extension of an ordinary Petri net to a coloured Petri net provides the ability to represent different types of token and treat them differently. In a coloured Petri net, the transition is only enabled by an incoming token if the token's colour matches the colours allowed by the transition. A coloured Petri net can always be converted to an ordinary Petri net through additional places and transitions for each different colour.

The compass Petri net contains a number of different token types in order to represent the information types required within the compass program, such as the ASCII characters read in from the compass itself, or the final format of the data when a bearing has been established. However, these additional token types do not affect the choices made in this Petri net, and are therefore irrelevant to its analysis. In the case of the compass Petri net shown in Figure 7, there are essentially only two colors required in the Petri net, "pass" and "fail", as these are the only two colours relevant to the testing of the Petri net in simulation. Any other variations in token type serve no purpose in simulation but to increase the complexity of the analysis.

For the purposes of the calculations to be performed using Gröbner bases, the "pass" tokens have been labelled x, and the "fail" tokens y. The initial marking of the Petri net is described by a single "pass" token in the "start" place (1), and "pass" and "fail" tokens in each of the places "input" (18) and "continue" (19). The additional tokens at places 18 and 19 allow the user to perform the more rigorous testing of the coloured Petri net.

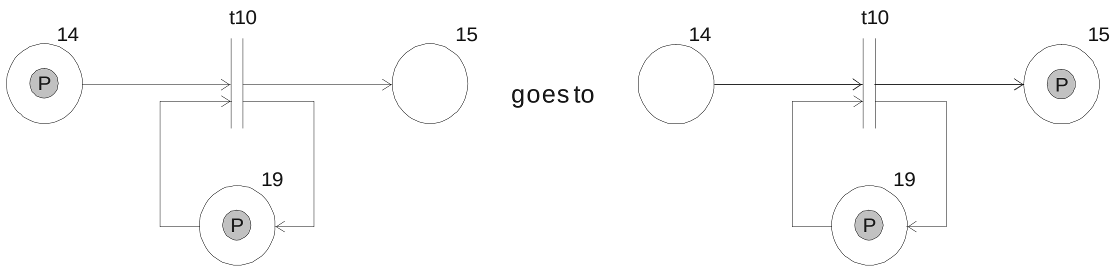
The "pass" and "fail" tokens are primarily used as a distinction between data received with a correct checksum and data received with a failed checksum. This colouring is used to ensure that only uncorrupted data is used to calculate the bearing, which will later be output to the main navigation module of the mobile robot. Any failed data is instead sent to the "data request" transition, which can then provide a connection to the "dead reckoning plug-in module" not shown in this diagram. This can be seen through tracing the route of a "pass" and then a "fail" token through the transitions "read in" ($t_4$ or $t_{21}$), "calculate checksum" ($t_5$ or $t_{22}$) and "test" ($t_6$ or $t_{23}$) to the conclusion of the decision at the transitions "find bearing" ($t_7$) or "data request" ($t_{16}$).

As shown in the previous example, the first step in the analysis of the Petri net and generation of the Grobner basis is to establish the polynomial for each transition. These polynomials are listed below, with a pass token represented by an x, and a fail token represented by a y.

For example, the polynomial:

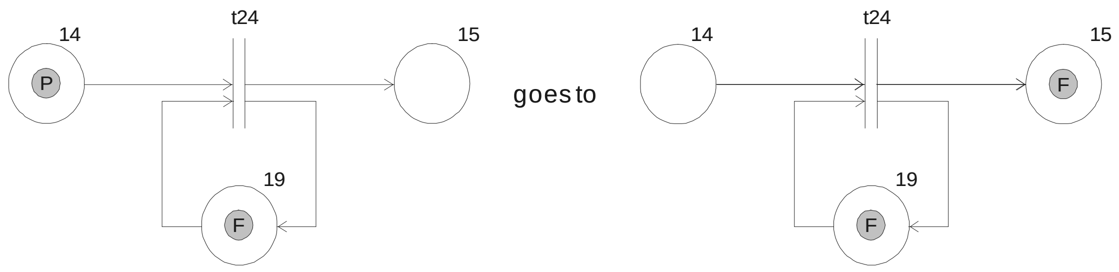
$$pol(t_{10}) = x_{14}x_{19} - x_{15}x_{19}$$

represents the following possible transition.



Note that pol($t_{24}$) is identical to pol($t_{10}$) apart from coloring.

$$\text{pol}(t_{24}) = x_{14}y_{19} - y_{15}y_{19}$$



Polynomials can be generated for the other transitions as follows.

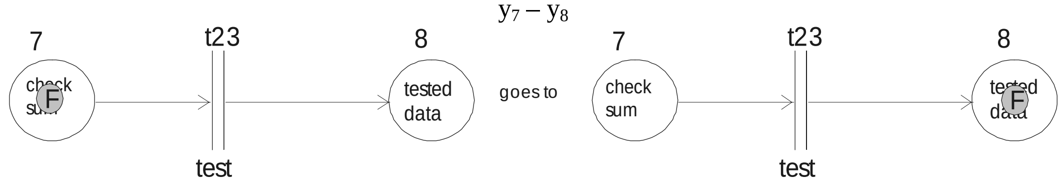
| | | |
|---|---|---|
| pol($t_3$) = $x_2x_{18} - x_3x_{18}$, | pol($t_1$) = $x_1$ - $x_2x_4$ | pol($t_{13}$) = $x_2x_{17} - x_{16}$ |
| pol($t_{20}$) = $y_2y_{18} - y_3y_{18}$ | pol($t_2$) = $x_5 - x_{12}$ | pol($t_{14}$) = $x_{15} - x_{12}$ |
| pol($t_4$) = $x_3x_{13} - x_6$ | pol($t_7$) = $x_8 - x_{10}$ | pol($t_{15}$) = $x_4 - x_5$ |
| pol($t_{21}$) = $y_3y_{13} - y_6$ | pol($t_8$) = $x_{12} - x_{13}$ | pol($t_{16}$) = $y_8 - x_9$ |
| pol($t_5$) = $x_6 - x_7$ | pol($t_9$) = $x_{11} - x_2x_{14}$ | pol($t_{17}$) = $x_9 - x_{11}$ |
| pol($t_{22}$) = $y_6 - y_7$ | pol($t_{11}$) = $y_{15} - x_{17}$ | pol($t_{18}$) = $x_{10} - x_{11}$ |
| pol($t_6$) = $x_7 - x_8$ | pol($t_{12}$) = $x_3x_{17} - x_{16}$, | pol($t_{19}$) = $x_{16} - x_1$ |
| pol($t_{23}$) = $y_7 - y_8$ | pol($t_{25}$) = $y_3x_{17} - x_{16}$ | |

The complex Gröbner basis automatically generated from these polynomials is shown below. Some of the resulting expressions are shown graphically to clarify the meaning of the generated polynomials. The polynomials shown in bold represent expressions which would require tokens to pass through the entire Petri net more than once in order for one marking to be reachable from the other.

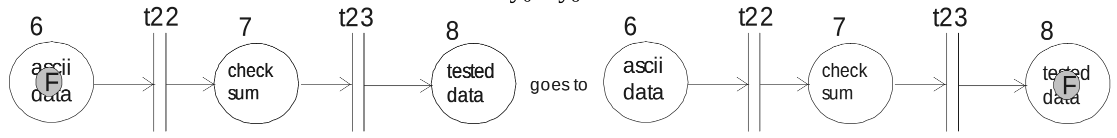
| | | |
|---|---|---|
| $y_3y_{15}y_{19} - y_8y_{19}$ | $x_2y_{18} - y_3y_{18}$ | $\mathbf{x_{14}x_3x_{18} - y_8x_{18}}$ |
| $y_{18}y_3y_{15} - y_8y_{18}$ | $x_2y_{15} - y_3y_{15}$ | $\mathbf{x_8 - y_8}$ |
| $x_{14}x_{19} - x_{15}x_{19}$ | $x_2x_{18} - x_3x_{18}$ | $\mathbf{x_7 - y_8}$ |
| $x_{16} - y_3y_{15}$ | $y_7 - y_8$ | $\mathbf{x_6 - y_8}$ |
| $y_3x_{15} - y_8$ | $y_6 - y_8$ | $\mathbf{x_{10} - y_8}$ |
| $y_{19}x_{15}x_{19} - x_{19}y_{15}y_{19}$ | $x_{17} - y_{15}$ | $\mathbf{x_{15}y_8 - y_{15}y_8}$ |
| $x_{14}y_8y_{18} - y_{15}y_8y_{18}$ | $x_{14}y_{19} - y_{15}y_{19}$ | $\mathbf{x_{14}y_{15}y_8 - y_{15}^2y_8}$ |
| $x_{14}y_8x_{18} - x_{18}y_{15}y_8$ | $\mathbf{x_2x_{15} - y_3y_{15}}$ | $\mathbf{x_{14}y_8^2 - y_{15}y_8^2}$ |
| $x_{14}y_3y_{15} - y_{15}y_8$ | $\mathbf{x_2x_{14} - y_8}$ | $\mathbf{x_{14}y_3y_{18} - y_8y_{18}}$ |
| $x_{13} - x_{15}$ | $\mathbf{x_1 - y_3y_{15}}$ | $\mathbf{y_3y_{15}^2 - y_{15}y_8}$ |
| $x_{12} - x_{15}$ | $\mathbf{x_2y_8 - y_3^2y_{15}}$ | $\mathbf{y_3y_{15}y_8 - y_8^2}$ |
| $x_{11} - y_8$ | $\mathbf{x_3y_{15} - y_3y_{15}}$ | $\mathbf{y_3y_{15}y_{19} - x_{19}y_8}$ |
| $x_9 - y_8$ | $\mathbf{x_3y_8 - y_3y_8}$ | $\mathbf{x_{18}y_3y_{15} - y_8x_{18}x_{17}}$ |
| $x_5 - x_{15}$ | $\mathbf{y_{18}x_3x_{18} - x_{18}y_3y_{18}}$ | |
| $x_4 - x_{15}$ | $\mathbf{x_3x_{15} - y_8}$ | |

As can be seen through the graphical representation of these first two polynomials, the polynomials which form the basis are not necessary in their simplest form, they show combinations of the polynomials formed by the transitions, with P representing a pass token, and F representing a fail token.

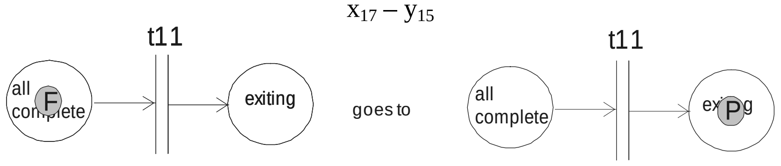
$$y_7 - y_8$$



$$y_6 - y_8$$

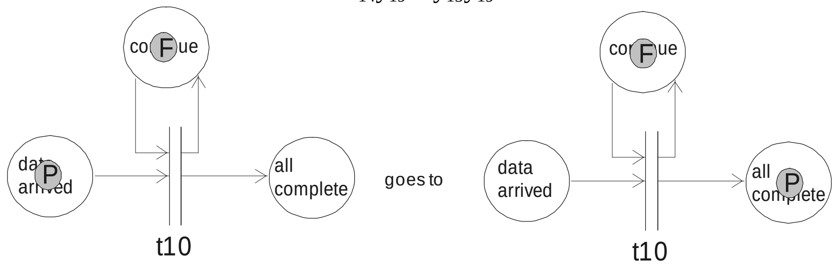


$$x_{17} - y_{15}$$



This diagram shows the change between a "fail" token and a "pass" token, possible only at specified transitions.

$$x_{14}y_{19} - y_{15}y_{19}$$



This diagram shows the token in the "continue" place (19) affecting the output of the transition. The equivalent polynomials for changes made through the "input" place (18) are shown in the first two polynomials below.

The remainder of the polynomials forming the Grobner basis for the compass Petri net (shown in bold) are more difficult to trace through the Petri net diagram, as they require tokens to pass around the Petri net more than once. This is perfectly acceptable, as the Petri net is expected to be reversible, but it does lead to polynomials which are misleading at first glance, and similarly to reachable markings where the path taken is unclear.

The shown completed Gröbner basis, once generated, can be used to find every reachable marking of the compass Petri net is represents. However, due to the length of this example, these results are not listed here.

Testing the reachability of this Petri net, given a certain type of token, will confirm that the choices made by colouring of a given token will behave as the user would expect, beyond the simple testing of a Petri net with no colourings. This will enable the user to determine errors of this nature prior to the final generation of executable code for the mobile robot and decrease the necessary debugging time.

## 6 Conclusions and Future Work

The method for Petri net analysis described here has proved highly reliable and accurate. It is particularly successful in the detection of Petri nets which have falsely been assumed to be reversible, and in finding badly designated initial markings of the Petri net which may also stop it from being reversible.

However, the time taken for performance of these calculations remained, as with many previous methods, unacceptably long despite the modularity of the model. The motors example shown in section 5 was completed successfully within a few minutes, but once a colouring was added alongside a number of places and transitions for the compass example (section 5) the time taken increased beyond that which could be considered reasonable for the user to wait, Gröbner basis generation taking approximately an hour.

The primary concern of any further work on this technique must be the reduction of the time taken for results of reachability testing to become available to the user. There are three possible avenues of research available, which may lead to an appropriate reduction in complexity. The first may consider the reduction of the Petri net itself. Whilst the TRAMP method of separating objects into individual components has already greatly decreased the Petri net complexity [Pezze 95] [Caloini 98], it is clear that further effort must be put into this in order to provide a usable analysis service. This may be implemented through the use of standard Petri net reduction techniques [Murata 89].

The processing time may then be further reduced through the improved implementation of the Grobner basis techniques [Fröberg 97], which themselves have a number of efficiency algorithms which have yet to be utilised in these initial testing procedures.

A further alternative to the Petri net reduction and Gröbner basis efficiency techniques is the introduction of on-the-fly matrix generation method commonly applied to automata in order to improve the efficiency of the equivalent of reachability testing [Larsen 97]. If this method were to be applied to the Gröbner basis reachability test for the Petri net, then the overheads from large Petri nets would decrease dramatically.

A further desirable development may arise from the introduction of timings to the Petri net, which are already a widely used tool, and provide a valuable additional level of analysis to the Petri net.

**References**

[Buchberger 98] An Algorithmic Criterion for the Solvability of a System of Algebraic Equations, Buchberger B (translation Abramson M and Lumbert R). Grobner Bases and Applications. Proc. London Math Soc. Vol 251. 1998.

[Buchholz 92] A hierarchical View on GCSPNs and its Impact on Qualitative and Quantitative Analysis. Buchholz P. Journal of Distributed Computing, 1992, Vol 15, pp 207 – 224

[Caloini 98] A Technique for Designing Robotic Control Systems Based on Petri Nets. Caloini A, Magnani G, Pezze M. IEEE Transactions on Control Systems Technology, Vol 6, No 1, pp 72-87. 1998.

[Cavalieri 97] Impact of Fieldbus on Communication in Robotic Systems. Cavalieri S, DiStefano A, Mirabella O. IEEE Transactions on Robotics and Automation, 1997, Vol 13, No. 1, pp 30-48

[Chandler 99] Gröbner Basis Procedures for Testing Petri Nets. Chandler A, Heyworth A. UWB Math preprint 99.11. 1999

[Chandler 99a] An Object-Oriented Petri Net Toolkit for Mechatronic System Design. Chandler A. PhD Thesis, Lancaster University, Engineering Department, 1999.

[Fröberg 97] An Introduction to Gröbner Bases. Fröberg R. John Wiley and Sons, 1997.

[Holt 96] Algebraic Methods for Image Processing and Computer Vision. Holt RJ, Huang TS, Netravali AN. IEEE Transactions on Image Processing, Vol 5, No 6, pp 976-986. 1996.

[Jensen 97] Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use Volume 1. Jensen K. Spring-Verlag. 1997.

[Larsen 97] Efficient Verification of Real-Time Systems: compact data structure and state-space reduction. Larsen KG, Larsson F, Pettersson P, Yi W. Proceedings – Real-Time Systems Symposium, 1997.

[Mascaro 98] Hand-in-Glove Human-Machine Interface and Interactive Control: Task Process Modelling Using Dual Petri Nets. Mascaro S, Asada HH. Proceedings - IEEE International Conference on Robotics and Automation, 1998, Vol 2, pp 1289-1295

[Murata 89] Petri Nets: Properties, Analysis and Applications. Murata T. Proceedings of the IEEE, 1989, Vol 77, No. 4, pp 541 – 580

[Pezze 95] Graph Models for Reachability Analysis of Concurrent Programs. Pezze M, Taylor RN, Young M. ACM Transactions on Software Engineering, Vol 4, No 2, pp 171-213. 1995.

[Simon 98] Design and Analysis of Synchronisation for Real-Time Closed-Loops Control in Robotics. Simon D, Castaneda EC, Freedman P. IEEE Transactions on Control Systems Technology, 1998, Vol 6, No. 4, pp 445-461

[Suh 96] Design of a Supervisory Control System for Multiple Robotic Systems. Suh IH, Yeo HJ, Kim JH, Ryoo JS, Oh SR, Lee CW, Lee BH. IEEE International Conference on Intelligent Robots and Systems, 1996, Vol 1, pp 332-339

[Yavuz 99] Conceptual Design and Development of a Navigation System for a Mobile Robot. Yavuz H, Chandler AK, Bradshaw A, Seward DW. Proceedings of CACD 99 (International Workshop on Engineering Design), 1999, pp 65 - 80

## Appendix A: Gröbner Bases Definitions [Chandler 99]

Let X be a set. Then the elements of $X^{\Delta}$ are all power products of the elements of X, including an identity 1, with multiplication defined in the usual way. The commutativity condition is summarized by xy=yx for all $x, y \in X$. Let K be a field. Then the elements of $K[X^{\Delta}]$ are sums of K-multiples of elements of $X^{\Delta}$, with the operations of addition and multiplication defined in the natural way:
$\sum_i k_i m_i \sum_j l_j n_j = \sum_{i,j} k_i m_i n_j$ , for $k_i, l_j \in K$ and $m_i, n_j \in X^{\Delta}$

Let $P \subseteq K[X^{\Delta}]$. Equivalence modulo P is denoted $=_P$. We say that two polynomials are equivalent modulo P if their difference can be expressed in terms of P, i.e.

$$f =_P g \Leftrightarrow f - g = u_1p_1 + \dots + u_np_n \text{ for some } p_1, \dots , p_n \in P, u_1, \dots , u_n \in K[X^{\Delta}].$$

An admissible ordering on $X^{\Delta}$ is a relation > such that m>1 for all $1 \neq m \in X^{\Delta}$, and such that if m>n then um>un for all $u \in X^{\Delta}$. We will also require the well-ordering property: that there is no infinite sequence $m_1>m_2>\dots$ of power products $m_1, m_2, \dots$ of $X^{\Delta}$.

Let > be admissible well-ordering on $X^{\Delta}$. The leading term of a polynomial p is the power product occurring in p that is largest with respect to >, and is denoted LT(p). The leading coefficient of p is the coefficient of LT(p) and is denoted LC(p). A term t is said to occur in a polynomial p with coefficient k if t is a term of p. Reduction modulo P with respect to > is written $\rightarrow_P$ and defined as

$$f \rightarrow_P h \Leftrightarrow = f - kmp$$

where mLT(p) occurs in f with coefficient $k'$ and $k = k'(LC(p))^{-1}$ for $p \in P$.
A repeated sequence of reductions (the reflexive, transitive closure of $\rightarrow_P$) is denoted $\rightarrow^*_P$. The symmetric closure of this is denoted $\leftrightarrow^*_P$ coincide.

## The Buchberger Algorithm

In 1965 Buchberger invented the concept of a Gröbner basis [Buchberger 98]. If a set of polynomials Q is a Gröbner basis for P then we can use Q to determine whether two polynomials are equivalent modulo P. Formally:

i. $f =_P g \Leftrightarrow f =_Q g$ for all $f, g \in K[X^{\Delta}]$.

ii. For all $f \in K[X^{\Delta}]$ there exist $f_1, \dots f_n \in K[X^{\Delta}]$ such that $f \rightarrow P\ f_1 \rightarrow_Q \dots \rightarrow_Q f_n$ where $n \in N$ and $f_n$ is irreducible.

iii. $f =_P g \Leftrightarrow$ there exists $h \in K[X^{\Delta}]$: $f \rightarrow^*_Q h$ and $g \rightarrow^*_Q h$.

**Theorem (Reachability and Equivalence of a Polynomial)**

Let N̲ be a reversible Petri net with initial marking $M_0$. Define P:={pol(t): t∈T}. Then a marking M is reachable in N̲ if and only if $\text{pol}(M_0) =_P \text{pol}(M)$.

**Proof**

First suppose that M is reachable. Then there is a firing sequence $M_0 \rightarrow t_1\ M_1 \rightarrow t_2 \dots \rightarrow t_{n-1}\ M_{n-1} \rightarrow t_n$ M. Therefore there exist u1, … , un $\in X^{\Delta}$ such that
$\text{pol}(M_0) - u_1\text{pol}(t_1) = \text{pol}(M_1)$, $\text{pol}(M_1) - u_2\text{pol}(t_2) = \text{pol}(M_2)$, … , $\text{pol}(M_{n-1}) - u_n\text{pol}(t_n) = \text{pol}(M)$.
Therefore $\text{pol}(M_0) - \text{pol}(M) = u_1\text{pol}(t_1) + \dots + u_n\text{pol}(t_n)$. Hence $\text{pol}(M_0) =_P \text{pol}(M)$.

For the converse, suppose pol(M0) =P pol(M). Then there is a sequence

$$\text{pol}(M_0) = u_1l_1,\ u_1r_1 = u_2l_2,\ \dots,\ u_{n-1}r_{n-1} = u_nl_n,\ u_nr_n = \text{pol}(M).$$

Where $\text{pol}(t_1) = l_1 - r_1, \dots, \text{pol}(t_n) = l_n - r_n \in P$, and $u_1, \dots, u_n \in X^{\Delta}$. Note that $l_1, r_1, \dots, l_n, r_n \in X^{\Delta}$. Now recall that $M_0$ is a marking. Since $\text{pol}(M_0) = u_1l_1$, we can deduce that $t_1$ is enabled. Therefore there is a marking $M_1$ such that $M_0 \rightarrow t_1\ M_1$ and $\text{pol}(M_1) = \text{pol}(M_0) - u_1\text{pol}(t_1)$. By induction this implies that there are markings $M_2, \dots, M_n$ such that there is a firing sequence $M_0 \rightarrow t_1\ M_1 \rightarrow t_2 \dots \rightarrow t_n\ M_n = M$. Hence M is reachable in N̲.

**Corollary (Gröbner Bases Determine Reachability)**

Reachability in a reversible Petri net can be determined using a Gröbner basis.

**Remark (Catalogue of Reachable Markings)**

Recall that Gröbner bases techniques use an ordering on the power products. There is a one-one correspondence between power products and markings. We can begin to catalogue the markings in increasing order. Given a Gröbner basis for the polynomials of the transitions of a Petri net it can be determined whether each marking is reachable: if the power product reduces to the same irreducible power product as the initial marking then it is reachable. In this way the Gröbner basis can be used to build up reachable markings.

# Generating and Exploiting State Spaces of Object-Oriented Petri Nets

Milan Češka, Vladimír Janoušek, and Tomáš Vojnar

Department of Computer Science and Engineering, Brno University of Technology
Božetěchova 2, CZ–612 66 Brno, Czech Republic
e-mail: {ceska, janousek, vojnar}@dcse.fee.vutbr.cz

**Abstract.** The article describes several concepts establishing a basis for generating and exploiting state spaces of the object-oriented Petri nets (OOPNs) associated to the tool called PNtalk. The influence of identifiers of dynamically appearing and disappearing instances upon the state space explosion problem is explained. Methods of working with identifiers based on sophisticated naming rules and mechanisms for abstracting names are described and compared. Several optimizations of state space generating algorithms for the context of OOPNs are mentioned, as well. Finally, some possibilities of specifying properties of systems to be checked over the state spaces of their OOPN-based models are discussed.

## 1 Introduction

Current complex distributed applications require dealing with dynamically arising and disappearing objects which can communicate, synchronize their actions, and migrate among particular nodes of the distributed environment they are running in. Particularly, distributed operating systems, groupware allowing a concurrent work of several people on the same project, or applications exploiting the technology of agents or mobile agents can be listed as examples of the above-mentioned applications.

A language called PNtalk based on object-oriented Petri nets (OOPNs) [Jan98] has been developed at the DCSE, TU Brno in order to support modelling, investigating, and prototyping complex distributed object-oriented software systems. PNtalk supports intuitive modelling all the key features of these systems, such as object-orientedness, message sending, parallelism, and synchronisation. This is achieved through working with active objects encapsulating sets of processes described by Petri nets. Processes inside the objects communicate via a shared memory, while objects themselves communicate by message passing.

Simulation is one of the ways of examining systems modelled by OOPNs and it is already supported by a prototype version of a tool called PNtalk [ČJV97]. Models of software systems created in PNtalk should be usable as prototypes of these systems, as well. Currently we are working on new implementations of PNtalk in Prolog and in Smalltalk which should allow to run OOPN-based prototypes in a truly distributed way.

Although we have started with simulation and prototyping, this article concentrates on the first steps made towards exploiting formal analysis and verification methods in the context of OOPNs. This approach can be considered an alternative to simulation because although we are not always able to fully verify or analyse the behaviour of a system, even partial analysis or verification can reveal some errors which tend to be different from the ones found by simulation [Val98]. We believe that object-orientation should allow us to relatively easy extract the subsystems to be verified and to abstract their surroundings.

Among the different attitudes to performing formal analysis or verification, using state spaces appears to be the most straightforward way for the case of OOPNs. Methods based on state spaces are quite universal, can be almost fully automated, and allow a relatively easy implementation. There have been proposed many different ways of alleviating their main deficiency — the state explosion problem [Val98]. Some of these methods can be adapted and optimized for the context of OOPNs, as well. Apart from that, it is also necessary to solve some new problems accompanying state spaces of OOPNs as a formalism with dynamic instantiation, such as the problem how to efficiently deal with identifiers of dynamically appearing and disappearing instances.

When working with state spaces of OOPNs (or more generally of any formalism with dynamic instantiation of some kind of components) it is necessary to pay careful attention to treating identifiers of objects (or in general some other kind of dynamically appearing and disappearing instances). Otherwise, many unnecessary states can be generated and the state spaces can even unnecessarily grow to infinity. This naming problem can be solved either by introducing some sophisticated rules for assigning identifiers to instances or by not considering concrete names of instances to be important when testing states to be equal.

The work with instance identifiers influences not only generating state spaces of OOPNs, but also analyzing them. This is because we need to be able to describe expected properties of the systems being examined without referring to the concrete names of the instances involved in states and events of their state spaces. These names are semantically not important and, what is more, modellers can hardly work out what identifiers will be used in different states.

In the article, we first present the main ideas behind the OOPN formalism. Subsequently we discuss possible solutions of the naming problem arising in state spaces of OOPNs, together with some further optimizations to be used when generating these state spaces. Finally, we suggest an approach to specifying properties to be evaluated over states spaces of OOPNs.

## 2 Key Concepts of OOPNs

The OOPN formalism [ČJV97] is characterized by a Smalltalk-based object-orientation enriched with concurrency and polymorphic transition execution, which allow message sending, waiting for and accepting responses, creating new objects, and performing primitive computations. An example demonstrating the notation of OOPNs is shown in figure 1.

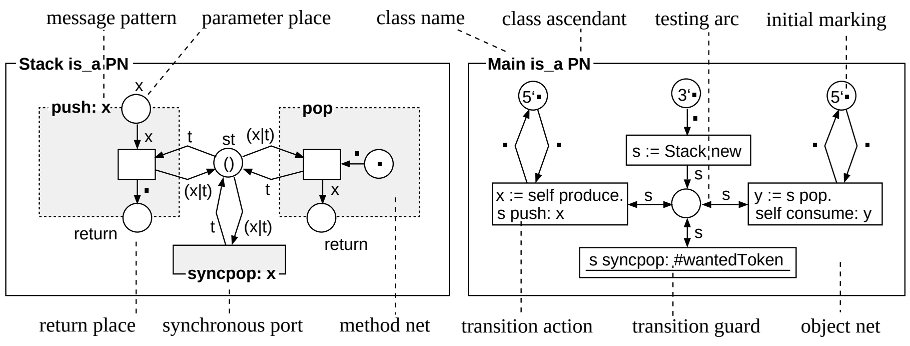


**Fig. 1.** An OOPN example (`Main`'s methods `produce` and `consume` are not shown).

This section rephrases the basic ideas of the definition of OOPNs, however, due to space limitations, without making the description formal and complete. We explain the necessary notions only. A bit deeper introduction to the OOPN formalism can be found in [ČJV97] and the entire definition of OOPNs in [Jan98].

## 2.1 The Structure of OOPNs

An *object-oriented Petri net* is a triple $(\Sigma, c_0, oid_0)$ where $\Sigma$ is a system of classes, $c_0$ an initial class, and $oid_0$ the name of an initial object from $c_0$.

$\Sigma$ contains sets of OOPN elements which constitute classes. It comprises constants $CONST$, variables $VAR$, net elements (such as places $P$ and transitions $T$), class elements (such as object nets $ONET$, method nets $MNET$, synchronous ports $SYNC$, and message selectors $MSG$), classes $CLASS$, object identifiers $OID$, and method net instance identifiers $MID$. We denote $NET = ONET \cup MNET$ and $ID = OID \cup MID$. The universe $U$ of an OOPN contains (nested) tuples of constants, classes, and object identifiers. Let $BIND = \{b \mid b : VAR \longrightarrow U\}$ be the set of all bindings of variables.

*Object nets* consist of places and transitions. Every place has some initial marking. Every transition has conditions (i.e. inscribed testing arcs), preconditions (i.e. inscribed input arcs), a guard, an action, and postconditions (i.e. inscribed output arcs). *Method nets* are similar to object nets but, in addition, each of them has a set of parameter places and a return place. Method nets can access places of the appropriate object nets in order to allow running methods to modify states of objects which they are running in.

*Synchronous ports* are special transitions which cannot fire alone but only dynamically fused to some other transitions which "activate" them from their guards via message sending. Every synchronous port embodies a set of conditions, preconditions, and postconditions over places of the appropriate object net, and further a guard, and a set of parameters. Parameters of an activated port $s$ can be bound to constants or unified with variables defined on the level of the transition or port that activated $s$.

A *class* is specified by an object net (an element of $ONET$), a set of method nets (a subset of $MNET$), a set of synchronous ports (a subset of $SYNC$), and a set of message selectors (a subset of $MSG$) corresponding to its method nets and ports. Object nets describe possible independent activities of particular objects, method nets reactions of objects to messages sent to them from outside, and ports allow to remotely test and change states of objects in an atomic way. The inheritance mechanism of OOPNs allows an incremental specification of classes. Inherited methods and synchronous ports can be redefined and new methods and synchronous ports can be added. A similar mechanism applies for object net places and transitions.

### 2.2 The Dynamic Behaviour of OOPNs

The dynamic behaviour of OOPNs corresponds to the evolution of a system of objects. An *object* is a system of net instances which contains exactly one instance of the appropriate object net and a set of currently running instances of method nets. Every *net instance* entails its identifier $id \in ID$ and a marking of its places and transitions. A *marking of a place* is a multiset of elements of the universe $U$. A *transition marking* is a set of invocations. Every *invocation* contains an identifier $id \in MID$ of the invoked net instance and a stored binding $b \in BIND$ of the input variables of the appropriate transition.

A state of a running OOPN has the form of a *marking.* To allow the classical Petri net-way of manipulating markings, they are represented as multisets of token elements. In the case of a transition marking, the identifier of the invoked method net instance is stored within the appropriate binding in a special (user-invisible) variable `mid`. Thus a formal compatibility of place and transition markings is achieved and it is possible to define a token element as a triple consisting of the identifier of the net instance it belongs to, the appropriate place or transition, and an element of the universe or a binding. Then we can say for a marking $M$ that:

$$M \in [(ID \times P \times U) \cup (ID \times T \times BIND)]^{MS}.$$

A step from a marking of an OOPN into another marking can be described as the so-called *event.* Such an event is a 4-tuple

$$E = (e, id, t, b)$$

including (1) its type $e$, (2) the identifier $id \in ID$ of the net instance it takes place in, (3) the transition $t \in T$ it is statically represented by, and (4) the binding tree $b$ containing the bindings used on the level of the invoked transition as well as within all the synchronous ports (possibly indirectly) activated from that transition. There are four kinds of events according to the way of evaluating the action of the appropriate transition: `A` – an atomic action involving trivial computations only, `N` – a new object instantiation via the message `new`, `F` – an instantiation of a Petri-net described method, and `J` – terminating a method net instance. If an enabled event $E$ occurs in a marking $M$ and changes it into a marking $M'$, we call this a *step* and denote it by

$$M[E\rangle M'.$$

For a given OOPN, its *initial marking* $M_0$ corresponds to a single, initially marked object net instance from the initial class $c_0$ identified as $oid_0$. The *set of all markings* of an OOPN is denoted as $MA$ and the *set of all events* as $EV$.

Finally, we introduce the following notation. Given $id \in ID$, $net(id)$ denotes the net $n \in NET$ such that $id$ identifies an instance of $n$, and $oid(id)$ denotes $oid \in OID$ such that $id$ identifies a net instance belonging to the object identified by $oid$. Note that an object is identified by the identifier of its object net instance.

## 3 The Role of Instance Identifiers in State Spaces of OOPNs

In this section there is identified a special problem arising in the context of state spaces of OOPNs, namely the so-called naming problem associated to dealing with identifiers of instances that can be dynamically created and discarded. There are suggested and compared two methods trying to minimize the impact of the presence of names of instances in states upon the state space explosion problem. From the point of view of the naming problem, OOPNs can be considered just a representative of formalisms with dynamic instantiation of some kind of components, such as objects or processes.

### 3.1 The Naming Problem

State spaces can generally be defined [Val98] as 4-tuples consisting of a set of states, a set of structural transitions, a set of semantic transitions (i.e. links between states and structural transitions), and an initial state. This concept can be used when dealing with state spaces of OOPNs (or any other formalism with dynamic instantiation), as well. However, in the context of such formalisms there arises one new phenomenon to be considered. Particularly, it is necessary to pay careful attention to efficiently handling the naming information present in states in order not to worsen or even to decrease the state space explosion problem. Let us denote this phenomenon as the *naming problem*.

The naming information present in states of dynamically structured models serves for uniquely identifying the just existing instances of different model components which allows to separate their local states and express references among them. Working with instance identifiers, e.g. in the form of addresses of objects, is common when running object-oriented programs or simulating object-oriented models. However, in the context of state spaces, the presence of the naming information in states can significantly contribute to the state space explosion problem. This is due to the possibility of unnecessarily generating many states differing only in the names of the involved instances even if the names cannot influence the future behaviour of the system being examined in any way (up to renaming). What is worse, sometimes the naming information can make state spaces of evidently finite-state systems grow to infinity — it suffices to keep creating and destroying the same instance (from a semantical point of view) identifying it using still new identifiers.

State space redundancies associated to working with the naming information have two possible sources. The first one is a specialty of formalisms supporting dynamic structuring of some kind and stems from the way these formalisms *assign identifiers to newly arising instances*. Redundant states arise when an instance playing a certain role in the modelled system (given by its type, the local states it can reach, or the way it is referred to) can be created under different names when going through different state space paths leading to its creation. To illustrate such a case, we can consider for example a model of a flexible manufacturing system [JV98] with several production cells represented by objects uniquely distinguished by the machines they encapsulate. Redundant states are generated when these objects can be created and identified using different combinations of names, possibly reflecting the order in which they were independently of each other created. A similar situation often arises when a method can be invoked with the same arguments over the same object via different state space paths leading to different internal identifiers of the resulting instance.

The second possible source of redundancies associated to identifiers is concurrent, symmetrical work with several instances. In this case, some of the symmetries existing already on the level of the system under investigation are mapped onto treating certain instances via their identifiers in a symmetrical way. This can cause that we would not loose any information and would save some space if we could ignore the different identifiers of these instances and swap them in some points of the evolution of the modeled system. As an example, we can take the symmetrical work with dining (distributed) philosophers in figure 2. The possibility of exploiting system-level symmetries for reducing state spaces is not a specialty of formalisms with dynamic instantiation, but it seems to be interesting that at least some of these symmetries manifest themselves in a similar way to redundancies stemming purely from some internal mechanisms of dynamic structuring and might thus hopefully be solved together.

The naming problem can be solved either by restricting the semantics of the applied modelling language or by reducing the resulting state spaces appropriately. We prefer the second approach which allows to use different methods for solving the naming problem in different contexts and to combine them with methods trying to remove other sources of state space redundancies. When reducing state spaces to project away what we consider to be unnecessary naming information we have to prove that doing this we do not lose anything or at least anything important. Such a proof is to be based on specifying what information is considered important from the point of view of using state spaces for formal analysis and verification. What is more, even if we do not lose any information when solving the naming problem, the notion of important information can be used for showing that such information can be extracted from the appropriately reduced state spaces in an efficient way. (In the context of OOPNs, it is normally important to be able to distinguish particular instances in states, to express their ownership relations, and to know how they are manipulated by events surrounding particular states. However, we are usually not interested in the concrete values of identifiers used for implementing the just mentioned mechanisms.)

In this section there are suggested and compared two methods for solving the naming problem: (1) using sophisticated naming rules for assigning identifiers to newly arising instances inspired by the way of handling process identifiers in Spin [Hol97] and (2) the so-called name abstraction as a specialization of the symmetry method for reducing state spaces [Jen94,CDFH97]. The latter method is based on not considering concrete names of instances to be important when checking states to be equal, which leads to working with renaming equivalence classes of states rather than with the individual states. Both of these methods together with their pros and cons are discussed in the following in the context of OOPNs. However, first of all, we define full state spaces of OOPNs in order to obtain a basis to be reduced using one of the mentioned methods.

Still before discussing the two mentioned principles in more detail we should note that the problem they should solve cannot be avoided by simply presenting an algorithm for transforming the formalism with dynamic instantiation under question into some kind of low-level formalism which should serve as a basis for formal analysis. When we for example try to transform object-oriented Petri nets into some kind of "plain" high-level nets, as for example in [SB94], there must appear a construction generating identifiers which then become a distinguishing part of tuples representing tokens of originally different net instances folded together. Thus the problem of naming is carried into the domain of non-object nets and must be solved within their analysis process.

### 3.2 Full State Spaces of OOPNs

Full state spaces of OOPNs can be defined using the general concept of state spaces mentioned above. For a given OOPN, states will correspond to reachable markings and structural transitions to applicable events. Semantic transitions will be defined in accordance to the firing rules of OOPNs. Finally, the initial state will be the initial marking, of course.

**Definition 1 (Full State Spaces of OOPNs).**
*Let an object-oriented Petri net $OOPN$ with its set of markings $MA$, its initial marking $M_0$, and its set of events $EV$ be given. We define the* (full) state space *of $OOPN$ to be the 4-tuple $StSp = (S, T, \Delta, M_0)$ such that:*

1. $S = [M_0\rangle$.
2. $T = \{(M_1, E, M_2) \in S \times EV \times S \mid M_1[E\rangle M_2\}$.
3. $\forall M_1, M_2 \in MA\ \forall E \in EV[(M_1, E, M_2) \in T \Leftrightarrow (M_1, (M_1, E, M_2), M_2) \in \Delta]$.

A consequence of the definition of full state spaces of OOPNs ignoring the naming problem is that when we try to create the first instance of some net whose domain of its instance identifiers is infinite we immediately obtain infinitely many possible target markings. Moreover, requiring sets of possible identifiers of nets to be finite will not solve the problem because (1) it can change the semantics of the model by artificially restricting the number of concurrently existing instances and (2) there can still be generated unnecessarily many target markings.

## 3.3 Using Sophisticated Naming Rules

Sophisticated rules for assigning identifiers to newly arising instances attempt to decrease the degree of nondeterminism potentially present in the management of names of dynamically arising and disappearing instances and thus to decrease the number of reachable states.

The simplest nontrivial rule for naming instances is assigning identifiers according to some ordering over them. A deficiency of this attitude is that when we are cyclically creating and destroying some instance we will obtain an infinite state space. This can be solved by recycling identifiers immediately after they are released, i.e. by identifying newly arising instances by the smallest and currently not used identifiers. However, even then there can be generated many different states which are obviously semantically equal. Such a situation arises when some configuration of instances characterized by the number of the involved instances, their types, their trivial marking, and their mutual relations can be reached via several state space paths in which the instances are created in different orders and using various auxiliary instances with differently overlapped lifetimes. Then there can be generated several states containing the given configuration of instances and differing only in the names of some of the involved uninterchangeable instances (distinguished by their contents or by the way they are referred to).

The just described problem of generating unnecessary states can be decreased by using mutually independent sequences of identifiers for each type of instance. These sequences must, of course, be disjoint, which can be achieved for example via making types of instances a part of their instance identifiers. In such a case, if we create the first instance of a net $n_1$ and then the first instance of a net $n_2$, they will not be identified 1 and 2 (which would undesirably reflect the order in which the instances were created), but for example $(n_1, 1)$ and $(n_2, 1)$. After such an optimization naming redundancies arise only if there appear different orders of creating instances of the same type, which can be less frequent.

A similar principle to the above can be used to achieve a further reduction of naming redundancies in the area of identifying method net instances. Here, we can keep assigning identifiers to method net instances independently for each transition and its input part binding used within some `F` event. This is possible due to each method net instance is referenced only from the marking of the transition which started it and the appropriate marking element contains the binding of the transition used when starting the method. However, it is again necessary so that the different sequences of potential identifiers are disjoint.

To illustrate generating how many unnecessary states we can avoid just via adding the last optimization above to the previous ones, we can use the following data. In the case of the system of 3 distributed philosophers from figure 2, we obtain 8127 states instead of 79292 ones. Similarly, for a simple system of 3 restartable cyclic counters from [Voj00], we obtain 34237 states instead of 66151. However, we shall note that even when applying all the above heuristics there can still remain some redundancies. Generally, these redundancies stem from (1) the mechanisms of assigning names to objects from the same classes and to method net instances started under the same binding of the same transition and from (2) mapping system-level symmetries (i.e. symmetries existing already on the level of modeled systems) onto working with instance identifiers.

The problem of generating unnecessary states, which can hardly be avoided even under advanced naming schemes, can be alleviated to some degree when using partial order reduction techniques (also known as commutativity-based methods) [Val98]. This is because these techniques reduce numbers of paths leading to particular states and thus also possibilities to obtain different permutations of identifiers of the involved characteristic instances. Nevertheless, the problem is not fully solved this way as it is not always possible to choose only one interleaving out of a set of the possible ones. Partial order techniques can ignore different orders of actions only in the case they are invisible (w.r.t. the property being checked) and do not collide. Furthermore, finding optimal stubborn sets can be too time-consuming and so an approximation is often taken (especially in the case of high-level formalisms). Finally, so far we have only considered the first possible source of redundancies associated to instance identifiers — namely assigning identifiers to instances being created. However, there is still the possibility of mapping system-level symmetries onto symmetrical work with some instances via their identifiers, which makes it possible to reduce state spaces by swapping the roles of such instances (thus ignoring their different identifiers) in some points of the evolution of the systems being examined. Unfortunately, sophisticated naming rules, even when combined with partial-order reduction methods, cannot provide such a reduction.

A certain kind of sophisticated naming rules similar to the above has originally been used for solving the naming problem in the already mentioned state space tool Spin [Hol97]. Spin works with dynamically instantiable processes described in a specialized language Promela. Its processes are identified by integers and can be terminated and their identifiers recycled only in the reverse order to their creation starting with the "youngest" currently running process. This principle can be easily implemented, but, on the other hand, if there is a possibility of cyclic instantiation of processes with partially overlapped lifetimes, the method will not prevent the state space from growing to infinity. (Of course, if we do not apply artificial limits on the number of running processes.) Moreover, the fact that identifiers can be recycled only in the reverse order to their allocation is acceptable in the context of processes, but it is less suited when working with objects whose life is often independent of their creators.

### 3.4 Abstracting Away the Naming Information

With respect to the previous discussion, we now suggest another possible method for solving the naming problem based on not considering concrete values of names of instances to be important when checking states to be equal. In other words, we are going to define two markings to be equal if there exists a suitable permutation over the set of all identifiers whose application makes the states identical. As a consequence, we will replace working with particular states by working with renaming equivalence classes of them. In the following, we will try to describe the method at least partially in a formal way — a fully formal description, together with proofs of the propositions, can be found in [Voj00].

It should be noted here that the concept of name abstraction is a specialization of the general notion of symmetries [Jen94] applied for reducing state explosion caused by the presence of concrete names of instances in states. Unlike general symmetries, highly specialised renaming symmetries can be used within all OOPN-based models and allow fully automated ways of treating them within generating state spaces. Name-abstracted state spaces could be described as a special case of symmetrically reduced state spaces, but we will define them straight here to save some space and get closer to their implementation.

The idea of abstracting away the naming information can only be applied due to the fact that the behaviour of OOPN-based models does not depend on concrete values of identifiers. Here, it is crucial that the definition of OOPNs does not allow to use instance identifiers in expressions and that there cannot be performed trivial computations depending on concrete values of instance names. Therefore it can be proved that starting from some state concrete names of instances do not influence the future evolution of the appropriate OOPN in any way (up to renaming). Thus it is not necessary to distinguish states equal up to renaming because of the future behaviour they can lead to.

We have said that we want two markings to be equal if there exists a suitable permutation over the set of all the identifiers allowed in the appropriate OOPN whose application makes the states identical. However, we do not accept all permutations. An acceptable permutation must preserve the information about (1) to which object a given instance belongs to, (2) to which net the instance belongs, and (3) it cannot change the identifier of the initial object, which is important for the garbage collecting mechanism. Permutations conform to the just described requirements will be called *renaming permutations* in the following.

**Definition 2 (Renaming Permutations).**
*Suppose we have an object-oriented Petri net $OOPN$ with its set of instance identifiers $ID$ and its initial object identifier $oid_0$. We define renaming permutations over $OOPN$ to be the bijections $\pi : ID \leftrightarrow ID$ such that:*

1. $\pi(oid_0) = oid_0$.
2. $\forall id \in ID \; [net(id) = net(\pi(id))]$.
3. $\forall id \in ID \; [\pi(oid(id)) = oid(\pi(id))]$.

The concept of renaming permutations provides a basis for defining the so-called *renaming symmetries*, i.e. bijections on sets of markings and sets of events. The formal definition of renaming symmetries can be obtained by a simple but a little longer extension of bijections working over identifiers to bijections over markings and events — we will skip the definition here. We denote the renaming symmetry induced by a renaming permutation $\pi$ as $\varrho^\pi$. Now we can define two markings $M_1$, $M_2$ to be *equal up to renaming* iff there exists a renaming permutation $\pi$ such that $\varrho^\pi(M_1) = M_2$. The same can be done for events. In the following, we will denote the renaming equivalence relation by $\sim$. Members of its equivalence classes will be referred to using the black board alphabet, i.e. $\mathbb{M}$ or $\mathbb{E}$, or via their representatives, i.e. $[M]$ or $[E]$. Finally, quotient sets w.r.t. $\sim$ will be denoted using $\sim$ as a subscription, as e.g. $MA_\sim$.

The notion of renaming symmetries allows us to formalize the already mentioned proposition that concrete names of instances cannot influence anything else than again names of instances present in the future behaviour of an OOPN-described system starting from a given state. Such a property is crucial in the theory of symmetrically reduced state spaces.

**Proposition 1.** *Let us have an object-oriented Petri net OOPN with its set of markings $MA$, its set of events $EV$, and the corresponding set of all renaming permutations $\Pi$. Then the following holds for every $M_1, M_2 \in MA$, $E \in EV$, and $\pi \in \Pi$: $M_1[E\rangle M_2 \Leftrightarrow \varrho^\pi(M_1)[\varrho^\pi(E)\rangle\varrho^\pi(M_2)$.*

Renaming symmetries allow us to propose the expected notion of *name-abstracted state spaces (NA state spaces)* of OOPNs. When generating a name-abstracted state space, concrete identifiers of instances will not be taken into account and two states or events will be considered equal if they are equal up to renaming. The definition of name-abstracted state spaces will be based again on the general concept of state spaces. However, this time states will correspond to reachable *name-abstracted markings*, i.e. equivalence classes of $MA$ w.r.t. $\sim$, and structural transitions to useful *name-abstracted events*, i.e. equivalence classes of $EV$ w.r.t. $\sim$. Semantic transitions will be defined in accordance to the firing rules of OOPNs and to the semantics of renaming. The initial state will be equal to the equivalence class comprising the initial marking and only the initial marking.

**Definition 3 (Name-Abstracted State Spaces).**
*Let an object-oriented Petri net OOPN with its set of markings MA, its initial marking $M_0$, its set of events $EV$, and its renaming equivalence relation $\sim$ be given. We define the* name-abstracted state space (NA state space) *of OOPN to be the 4-tuple $NAStSp = (S_N, T_N, \Delta_N, [M_0])$ such that:*

1. $S_N = \{[M] \in MA_\sim \mid M \in [M_0\rangle\}$.
2. $T_N = \{([M_1], [E], [M_2]) \in S_N \times EV_\sim \times S_N \mid M_1[E\rangle M_2\}$.
3. $\forall \mathbb{M}_1, \mathbb{M}_2 \in MA_\sim\ \forall \mathbb{E} \in EV_\sim\ [(\mathbb{M}_1, \mathbb{E}, \mathbb{M}_2) \in T_N \Leftrightarrow (\mathbb{M}_1, (\mathbb{M}_1, \mathbb{E}, \mathbb{M}_2), \mathbb{M}_2) \in \Delta_N]$.

The above proposed NA state spaces are based on projecting away the naming information present in particular states and structural transitions of the classical state spaces. NA state spaces preserve information about reachable states and events, but their interconnection is preserved only partially. This fact is formalized in proposition 2 from whose structure we can guess that NA state spaces do not contain information about which particular instances are manipulated by events when going from one state into another.

**Proposition 2.** *Let us have an OOPN with its state space StSp and the corresponding name-abstracted state space $NAStSp$. Then the following holds:*
$\forall n \geq 1\ \forall \mathbb{M}_1, \ldots, \mathbb{M}_n \in MA_\sim\ \forall \mathbb{E}_1, \ldots, \mathbb{E}_{n-1} \in EV_\sim\ \forall i \in \{1, \ldots, n\}\ \forall M_i \in \mathbb{M}_i$ [ $\langle \mathbb{M}_1, \mathbb{E}_1, \ldots, \mathbb{M}_i, \ldots, \mathbb{E}_{n-1}, \mathbb{M}_n \rangle$ *is a path in NAStSp if and only if* $\exists M_1 \in \mathbb{M}_1, \ldots, M_{i-1} \in \mathbb{M}_{i-1}, M_{i+1} \in \mathbb{M}_{i+1}, \ldots, M_n \in \mathbb{M}_n\ \exists E_1 \in \mathbb{E}_1, \ldots, E_{n-1} \in \mathbb{E}_{n-1}$ *such that* $\langle M_1, E_1, \ldots, M_i, \ldots, E_{n-1}, M_n \rangle$ *is a path in StSp* ].

The information we are losing in NA state spaces is obviously not important when we are dealing with isolated states only. On the other hand, if we need to be able to explore sequences of states and events, this information might become necessary even if we do not consider concrete names of instances to be important. This is because it can be useful to know how a particular instance given by its identifier within some arbitrarily chosen representative of the appropriate NA state behaves within the events surrounding the state being examined. However, if we need the information, which is lost in NA state spaces, it is not difficult to preserve it. For this reason, we define the so-called *complete name-abstracted state spaces (CNA state spaces)*.

We define CNA state spaces as labelled NA state spaces. Every NA state will be labelled by a tuple consisting of a representative marking belonging to the equivalence class represented by the NA state and a set of self-renaming permutations (i.e. permutations which map the representative marking to itself). Every structural transition will be labelled by a set of tuples consisting of a representative event and a renaming permutation. We require that every representative event must be firable from the appropriate representative source marking leading to the appropriate representative target marking *after* applying the given renaming. Furthermore, every event firable from the representative source marking and leading to a marking equal up to renaming to the representative target marking must be derivable (up to the name of an eventually newly arising instance) from some of the representative events via a permutation from the set of source self-renaming permutations. Self-renaming permutations can decrease the number of target markings we have to process [Jen94].

**Definition 4 (Complete Name-Abstracted State Spaces).**
*Suppose we have an OOPN with its set of markings $MA$, its set of events $EV$, and its set of renaming permutations $\Pi$. We define the* complete name-abstracted state space (CNA state space) *of OOPN to be the triple* $CNAStSp = (NAStSp, m, e)$ *such that:*

1. $NAStSp = (S_N, T_N, \Delta_N, [M_0])$ *is the NA state space of OOPN.*
2. $m : S_N \to MA \times 2^{\Pi}$ *such that* $\forall \mathbb{M} \in S_N\ [m(\mathbb{M}) = (M, \Phi) \Rightarrow (M \in \mathbb{M}\ \wedge \forall \varphi \in \Phi\ [\varrho^{\varphi}(M) = M])]$.
3. $e : T_N \to 2^{EV \times \Pi}$ *such that for all* $(\mathbb{M}_1, \mathbb{E}, \mathbb{M}_2) \in T_N$ *with* $m(\mathbb{M}_1) = (M_1, \Phi_1)$ *and* $m(\mathbb{M}_2) = (M_2, \Phi_2)$, $e((\mathbb{M}_1, \mathbb{E}, \mathbb{M}_2))$ *is the smallest set such that* $\forall E' \in \mathbb{E}\ \forall M_2' \in \mathbb{M}_2\ [M_1[E'\rangle M_2' \Rightarrow \exists (E, \pi) \in e((\mathbb{M}_1, \mathbb{E}, \mathbb{M}_2))\ [M_1[E\rangle \varrho^{\pi}(M_2) \wedge \exists \varphi \in \Phi_1\ [eideq(E', \varrho^{\varphi}(E))]]]$.

In the definition of CNA state spaces, we have used a predicate *eideq* which is fulfilled when applied to two "existing identifier equal" events. Such events can differ only in the identifier of the newly created object within an `N` event and in the identifier of the newly started method net instance within an `F` event.

We should add that CNA state spaces are close to common symmetrically reduced state spaces [Jen94]. The use of *eideq* and not requiring all self-renaming permutations to be computed is an attempt to ease the automatic generation of CNA state spaces. (Note that it can be advantageous to try to obtain at least some self-renaming permutations starting by comparing events enabled in the same state.) CNA state spaces can be used with all models described by OOPNs. What is more, as we usually do not consider concrete names of instances to be important when analyzing properties of systems, there do not arise problems with the efficiency of extracting information from CNA state spaces even if the arising equivalence classes are infinite. Finally, CNA state spaces can be generated completely automatically, although some hints from modellers can sometimes improve the efficiency of generating them, as we will mention later.

CNA state spaces contain information about reachable states and events (up to renaming) and also about their concrete interconnection. This allows to find out how particular instances are manipulated by events. Consequently, it is possible to obtain a full state space from its CNA-variant and the set of all renaming permutations. These are the contents of the below proposition. Note that using the set of all renaming permutations is normally not necessary within practical verification tasks where concrete names of instances are not important.

**Proposition 3.** *Let us have an object-oriented Petri net $OOPN$ with its CNA state space $CNAStSp$ and its set of renaming permutations $\Pi$. Then it is possible to reconstruct the full state space of $OOPN$ from $CNAStSp$ and $\Pi$.*

The time complexity of generating CNA state spaces is almost the same as in the case of NA state spaces. This is because state representatives and renaming symmetries must be computed even when generating NA state spaces, although in their case they are thrown away after determining the target nodes of particular semantic transitions (thus saving some memory).

Let us now compare the attitudes of using sophisticated naming rules and name abstraction. We already know that sophisticated naming rules, possibly combined with partial order reduction, can remove some of the redundancies caused by assigning different names to dynamically arising instances with certain characteristic roles in the modeled system, but it is not guaranteed that they will remove all of them. Moreover, sophisticated naming rules, even when combined with partial order reduction, do not help much against system-level symmetries mapped onto working with certain instances in a symmetrical way via their identifiers. Name abstraction, on the other hand, can remove all the redundancy associated to names of dynamically appearing and disappearing instances and thus can save more memory than the attitudes based on sophisticated naming rules. This is a consequence of always ignoring all the different possibilities of identifying uninterchangeable instances within otherwise identical states without any respect to the way how they were created and what was their role so far.

So, it seems that renaming can save more memory than sophisticated naming rules, even in the case of using partial order reduction. On the other hand, we might have to pay for using renaming quite a lot in terms of the time complexity because testing markings to be equal up to renaming can multiply the overall time of generating state spaces by $O(n!)$ where $n$ is the maximal number of concurrently existing instances. Fortunately, this is the worst case scenario only and we can usually decrease the time complexity using the heuristics briefly mentioned in the next subsection. These heuristics are based on renaming insensitive hashing techniques decreasing numbers of states to be compared and on exploiting the structure of states for selecting instances whose identifiers it is sensible to permute. What is more, we can help the mechanism of name abstraction by manually specifying how to find and exploit renaming symmetries at least among some of the involved instances. In such a case, only the instances uncovered manually have to be processed by the further mentioned heuristics. It is also interesting that once we allow manual specification of renaming symmetries, the same mechanism can be used for specifying general symmetries and vice versa. Modellers can then freely choose how much information to provide manually and how much work should be done automatically. However, even in such a case, the above described principles of name abstraction should be respected.

To illustrate the above considerations, we can present some data obtained from a simple Prolog prototype of an OOPN state space generator, which we applied to several example models, such as classical philosophers, distributed philosophers, Russian philosophers, or different versions of simple systems with restartable counters [Voj00]. State spaces of these models obtained using the most elaborated sophisticated naming rules (without partial-order reduction) had about $4.10^2$–$4.10^4$ states. When using name abstraction they were reduced 1.1–414 times. From this, it is visible that name abstraction can really lead to significant reductions in numbers of states, although there exist systems (as e.g. the system of distributed philosophers) where already the most elaborated sophisticated rules remove most of the naming redundancies. The speedup obtained via using name abstraction in our examples ranged from very small negative values up to $2.10^3$. However, it is fair to note that the situation would change if we used better hash functions and if we were testing states to be identical via a straight comparison of the appropriate blocks of memory. On the other hand, using such techniques is generally problematic in tools based on functional or logical languages.

As a conclusion, we can say that more studies are still needed to answer the question whether and in which cases it is better to allow bigger memory consumption and when to use name abstraction.

### 3.5 Generating State Spaces of OOPNs

In this section, we will mention several methods intended to be used in the context of OOPNs to improve the efficiency of the classical algorithm of generating full state spaces [Jen94] as well as its plain or partial order reduced depth first variants typically used for formal verification [Pel96].

*Reducing Numbers of States to be Compared.* An important part of the time needed for generating a state space is spent testing whether a given state has already been included into the state space or not. In the case of NA state spaces, this is especially critical due to the time-consuming tests of equivalence up to renaming. We can reduce the time spent on comparing states by decreasing numbers of states to be compared. This can be achieved by representing state spaces by hash tables indexed via suitable hash functions working over states. Then we have to compare only the states for which the applied hash function returns the same value.

In the case of (C)NA state spaces, the employed hashing procedure must be insensitive to the mechanism of name abstraction applied here. To achieve this we suggest representing states of OOPNs as sequences of states of the just active objects, states of the active objects as sequences of states of the net instances encapsulated in the objects, and, finally, states of the net instances as sequences of the marking elements belonging to the particular instances. Before hashing on such state sequences, we further have to replace all the identifiers present in them by the names of the appropriate nets and sort the nested sequences of marking elements, instances, and objects according to the information they contain after the replacement of identifiers by the associated typing information. The described preparation of states for hashing is quite complex, however, it can fortunately be realized in an incremental way. The described scheme can be improved by replacing instance identifiers not only by the appropriate typing information, but also by some associative identification insensitive to renaming and easy to obtain. Associative identification of instances can be defined by modellers and can be based on summarizing what trivial objects are stored in what places of the instances and how these instances are referred to from the rest of the system. Finally, modellers can help the system by providing procedures for representing at least some critical parts of states in a unique way.

*Improving the Efficiency of Testing the Renaming Equivalence.* The worst case complexity of testing the renaming equivalence cannot be decreased, but the average one can be improved by exploiting the structure of states instead of blindly testing all permutations of identifiers. We can represent states as oriented graphs with two kinds of nodes corresponding to net instances and to marking elements. Instance nodes should be linked to the appropriate marking elements and those to the net instances they eventually contain. Testing states being equal up to renaming is then transformed to a unification of their *state graphs* treating identifiers as distinct typed variables. The unification can be implemented by synchronously traversing both state graphs and unifying their nodes. Doing this we should prefer unifying marking elements containing trivial objects or nontrivial objects with types which are unique within the marking of the appropriate places. Thus we can find differences as soon as possible and reduce the amount of backtracking stemming from matching objects of the same type stored in the same place with their different possible counterparts in the same place of the other state. Furthermore, we can exploit the already mentioned associative identification of instances here, as well. This can be done in such a way that we do not unnecessarily compare instances with different associative identification capturing their local states in a renaming insensitive way. Finally, modellers can again help the system by providing procedures for comparing at least some selected instances in a manually specified fast way.

*More Efficient Garbage Collecting.* The definition of OOPNs makes garbage collecting a part of every event. Performing garbage collecting means that we have to traverse the appropriate state graph (which can be computed incrementally) and find out which instances cannot be reached from the root. However, this computation is not necessary in every step because not every step makes some instance obsolete. It is sufficient to perform garbage collecting only when firing an event which includes a change of marking via arcs of a transition or a port inscribed by a *loss variable* bound to a nontrivial object. Loss variables of ports and transitions are the variables which appear on input but not on output arcs. Objects bound to such variables are likely to be lost and we have to check that using garbage collecting. It is not necessary to work with loss variables of transitions involved in `F` events because the complete binding of their variables is stored within the appropriate invocations in the marking of these transitions. In the case of `J` events, we have to start garbage collection also if there remain some nontrivial objects (different from the result of the event) in the net instance being finished. Runtime checking whether a loss variable is bound to a nontrivial object can sometimes be avoided using a static type analysis which can tell us that the appropriate variable can be bound to trivial objects only.

*Computing Enabled Events in an Incremental Way.* When we want to explore successors of a state we first have to compute the set of all the events enabled in that state. However, when we accept a little bit increased memory requirements we do not have to check the firability of all transitions in all net instances for every new state from a scratch. This is because the set of events enabled in a state can be computed in an incremental way starting with the set of events enabled in a predecessor state of the given state and just adding or removing some events according to re-checking the firability of some transitions. More precisely, we have to examine all the transitions which are connected to at least one input place whose marking was changed. Furthermore, we have to check transitions whose guards can use objects whose state was changed in a visible way. An object is changed in a visible way if there is a port in the class of the object which can read the contents of an object net place whose marking within the object was changed or which contains a visibly changed object. However, note that the just described mechanism can be applied only when we require that guard expressions of OOPNs cannot test any objects without or before reading them from some places. Nevertheless, this restriction of the notion of OOPNs seems to be quite practical.

## 4 Specifying Properties of Systems To Be Evaluated

In this section, we will discuss several possible ways of specifying properties to be evaluated over state spaces of systems modeled by OOPNs.

Most of the common ways of specifying properties to be checked over state spaces of models based on different modelling languages [Val98] can be used with OOPN-based models, too. We can think of using the following attitudes:

– evaluating *state space statistics* such as numbers of states, numbers of strongly connected components, bounds of places, Petri net live transitions, etc.,
– proposing a *versatile state space query language* to allow user-controlled traversing through state spaces and examining the encountered states,
– *instrumenting models* by property labels such as end-state labels, progress labels, assertions, etc. or by property automata,
– using a *high level specification language* such as some temporal logic.

Most of the above listed attitudes have to be slightly accommodated for the context of OOPNs and their state spaces. For example, bounds of places of OOPNs should be computed separately for particular instances and then a maximum should be chosen, particular property labels should be joint in a suitable way either with places or transitions of OOPNs, etc. However, there arises one more general problem here which influences almost all of the mentioned attitudes (more precisely all of them up to state space statistics). This problem is querying particular states and events of OOPNs.

The main problem to be solved when querying states and events of OOPNs stems from the dynamism of OOPNs. We have to prepare tools for exploring properties of sets of states and events in a way which respects the fact that the sets of existing instances, their names and relations can be different in every encountered state and cannot be fully predicted. Therefore it is not possible to use as simple queries as e.g. in Design/CPN, such as "take the marking of some place $p$ from the static net instance unambiguously identified by $id$".

Within a prototype of an OOPN state space tool, we have suggested a solution of the above problem based on two groups of functions. First of all, we use the so-called *instance querying functions*. They allow us to begin with the unique initial object net instance or with sets of the just existing instances of certain nets given statically by their types. Subsequently, they allow to recursively derive sets of the net instances or constants straight or transitively referenced from the already known instances via the marking of some of their places or transitions. There also exists a function returning the set of method net instances just running over some given objects.

Instance querying functions are intended to be combined with the so-called *set iterating functions* in order to obtain the appropriate characteristics of states. Set iterating functions allow searching somehow interesting instances or constants in sets returned by the instance querying functions. We can for example take all the just existing instances of some net, select the ones which contain some constant in some place, and then go on by exploring some other instances referenced from the selected ones.

So far we have been speaking about functions for querying OOPN states only. However, examining events seems to be a little easier. It is enough to have tools for accessing the particular items of events, i.e. their type, the transition they are bound to, the instance they are firing in, and the appropriate binding.

The functions for querying states and events can be straight used as a part of a versatile OOPN state space query language for examining the encountered states and events. Moreover, they can be used for describing terms embedded in temporal logic formulae specifying properties of systems to be verified using their OOPN-based models. Finally, they can also be applied when specifying legal termination states, progress events, or system invariants.

We will now describe a little more some of the instance querying functions. We describe them in the form of Prolog predicates as they are declared in the prototype tool using them. They all take the current state to be implicit and return the result via their last parameter. The predicate `init(Is)` returns the set with the initial object net instance. The predicate `inst(Cs,Ns,Is)` returns the set of the just existing instances belonging to the nets from the set `Ns` and running over objects belonging to the classes from `Cs`. The predicate `token(Is,Ps,Cs,Ms)` returns the set of tokens belonging to the classes from `Cs` and stored in the places from `Ps` within the instances from `Is`. The predicate `invoc(Is,Ts,Cs,Ns,Bs)` returns the set of invocations of the transitions from `Ts` within the instances from `Is`. The invocations are represented by the appropriate bindings and only the ones are selected which launch nets from `Ns` over objects of the classes from `Cs`. Finally, the predicate `over(Is1,Ns,Is2)` collects all the instances of the nets in `Ns` which run over the objects in `Is1`.

Out of the group of the set iterating functions, we can mention for example the following ones. The predicate `sforall(S,X,P,Y)` returns `true` in `Y` iff the predicate `P` over `X` is fulfilled over every element of the non-empty set `S` whose elements are one-by-one bound to `X`. Otherwise, a counter-example is found and bound to `Y`. The predicate `select(S1,X,P,S2)` selects all the elements `X` from `S1` which fulfill the predicate `P` over `X`.

Let us now present an example of examining states of OOPNs. For this reason, we will use a model of the system of distributed philosophers from [Jan98]. The class describing particular distributed philosophers is shown in figure 2. The whole model of the system of distributed philosophers contains yet another class whose only task is to create a group of philosophers and interconnect them into a ring via the methods `leftNb:` and `rightNb:`. Distributed philosophers differ from the classical ones in not having a shared table which could be used for exchanging forks. Instead they have to negotiate about forks via message sending using the methods `giveLFork` and `giveRFork`.

Below we define a Prolog predicate `eating_neighbours` allowing to derive the set of eating philosophers having an eating neighbour from a given state of the model of distributed philosophers. The current state is considered to be implicit. The predicate can be used to check the correctness of the proposed system which should not allow two neighbours to eat at the same time. Thus the predicate should always hold for the empty set only. This can be checked by a suitable state space query which evaluates the predicate `eating_neighbours` over every state and collects the states where it holds for a non-empty set. A more abstract approach would be checking the validity of the CTL formula $AG$ `eating_neighbours`($\emptyset$).

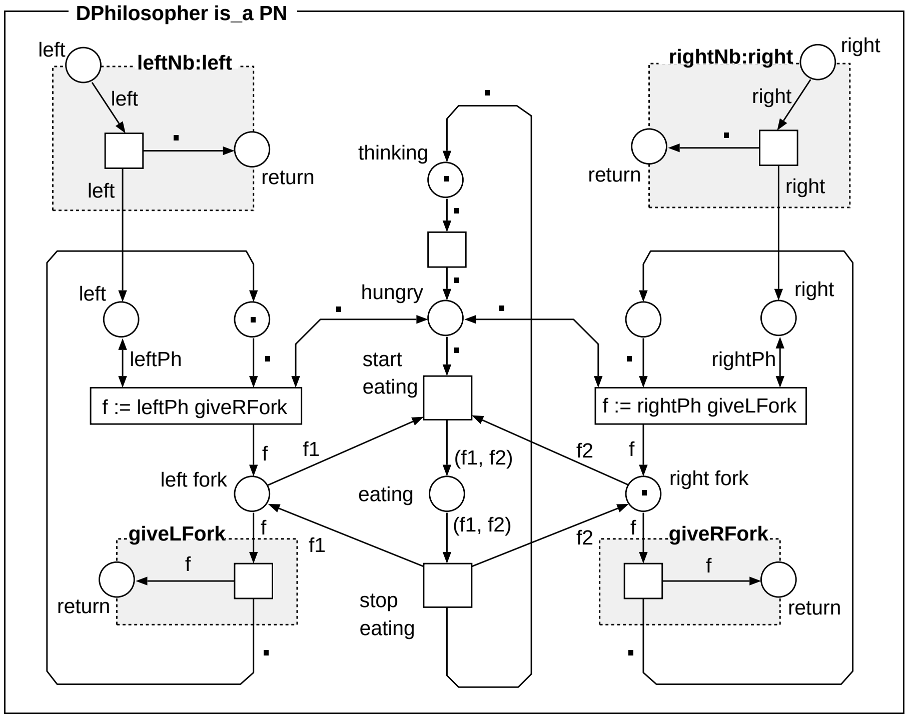


**Fig. 2.** The class of distributed philosophers from [Jan98]

```
eating_neighbours(EN) :-
  inst([dPhil],[[dPhil,object]], I),
  select(I,P,(token([P],[eating],all,E1),empty(E1,false),
              token([P],[left,right],all,LR),
              token(LR,[eating],all,E2),empty(E2,false)), EN).
```

## 5 Conclusions

In the article, we have briefly described the notion of object-oriented Petri nets associated to the tool called PNtalk and some of the problems accompanying generating their state spaces. We have especially mentioned the influence of working with identifiers of dynamically arising and disappearing net instances upon the state space explosion. Two possible approaches of dealing with the identifiers, namely sophisticated naming rules and name abstraction, have been described and compared. Some further optimizations of generating state spaces of OOPNs have been mentioned, as well. The methods for optimizing garbage collection and for computing enabled transitions in an incremental way despite their non-local influence can be used not only when generating state spaces of OOPNs, but also when simulating systems modeled by OOPNs.

We have discussed a method allowing to ask analysis or verification questions over OOPN state spaces, as well. This method avoids referring to uninteresting and unknown concrete names of instances and can be used within common ways of specifying properties to be evaluated.

The notions included in the article are supposed to be exploited within formal analysis and verification on suitably reduced OOPN state spaces which is one of the goals of our future research. As for reducing OOPNs' state spaces, we intend to adapt the theory of partial order reductions for the domain of OOPNs because there is quite a lot of concurrency in models based on them. We further intend to do more research on using OOPNs for modelling distributed systems, and especially the software ones.

**Acknowledgment.** This work was done within the research intention No. CEZ:J22/98: 262200012 - "Research in Information and Control Systems" and it was also supported by the Grant Agency of the Czech Republic under the contract 102/00/1017 "Modelling, Verifying, and Prototyping Distributed Applications Using Petri Nets".

## References

[CDFH97] G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad. A Symbolic Reachability Graph for Coloured Petri Nets. *Theoretical Computer Science*, 176:39–65, 1997.

[ČJV97] M. Češka, V. Janoušek, and T. Vojnar. PNtalk – A Computerized Tool for Object-Oriented Petri Nets Modelling. In F. Pichler and R. Moreno-Díaz, editors, *Proc. of EUROCAST'97.*, vol. 1333 of *Lecture Notes in Computer Science*, Las Palmas de Gran Canaria, Spain, 1997. Springer-Verlag.

[Hol97] G.J. Holzmann. The Model Checker Spin. *IEEE Transactions on Software Engineering*, 23(5), May 1997.

[Jan98] V. Janoušek. *Modelling Objects by Petri Nets.* PhD thesis, Faculty of Electrical Engineering and Computer Science, TU Brno, Czech Republic, 1998.

[Jen94] K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use, Vol. 2: Analysis Methods.* EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1994.

[JV98] V. Janoušek and T. Vojnar. Modelling a Flexible Manufacturing System. In J. Štefan, editor, *Proceedings of MOSIS '98, Vol. 2*, pages 195–200, Sv. Hostýn, Czech Republic, May 1998. MARQ Ostrava.

[Pel96] D. Peled. Combining Partial Order Reductions with On-the-fly Model-Checking. *Journal of Formal Methods in Systems Design*, 8 (1):39–64, 1996.

[SB94] C. Sibertin-Blanc. Cooperative Nets. In R. Valette, editor, *Proceedings of ICATPN'94*, volume 815 of *Lecture Notes in Computer Science*, pages 471–490, Zaragoza, Spain, June 1994. Springer-Verlag.

[Val98] A. Valmari. The State Explosion Problem. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, pages 429–528. Springer-Verlag, 1998.

[Voj00] T. Vojnar. *State Spaces of Object-Oriented Petri Nets.* PhD thesis, Faculty of Electrical Engineering and Computer Science, Brno University of Technology, Czech Republic, to be finished in 2000.

# The OCoN Approach for Object-Oriented Distributed Software Systems Modeling

Holger Giese and Guido Wirtz

Institut für Informatik, Westfälische Wilhelms-Universität Münster
Einsteinstraße 62, 48149 Münster, GERMANY
{gieseh,guidow}@math.uni-muenster.de

**Abstract.** The problems of todays software engineering for complex distributed software systems with control as well as data processing aspects are manifold. Besides the general problem of software complexity we additionally have to deal with the problems of concurrency and distribution. A set of well evolved formalisms especially w.r.t. concurrency exists, while their integration into the common software engineering framework is still missing and related attempts have often not gained the intended acceptance. But ever increasing system complexity as well as a fast growing market for distributed software effectuate a shift towards high level behavior modeling. The presented OCoN approach does provide a high level behavior modeling as extension to the UML de-facto standard for object-oriented modeling. It is an approach to integrate an adjusted Petri net formalism with the software engineering world.

## 1 Introduction

While place/transition nets [9] are accepted as one standard formalism of *software engineering* (cf. [42]) is the situation quite different for high-level Petri nets (*HLPN*). With the development of object-oriented analysis and design [44, 15] the shift towards a more high-level design view further boosts, but other behavioral formalisms like statecharts [28] win recognition. The high-level Petri net formalisms play no prominent role for object-oriented behavior modeling in practice.

There are several rational as well as historical reasons for this situation. A Petri net is a conceptional extension of an automaton and thus it is inherently more complex than state machines. The adequate handling of concurrency is still a complex problem and often solutions applying database technology that provides *parallel access transparency* to avoid the related problems is more appropriate. Thus, the development of complex software system engineering incorporating sophisticated concurrency aspects had been less influential and thus suitable concepts for object-oriented concurrent behavior modeling are not the main stream. Nowadays development of object-oriented methods and notations as well as the UML [40] neglect systems with concurrency.

It also has been detected that the expressiveness of the basic Petri net formalism is not sufficient to handle real modeling problems and thus several high-level extensions have been suggested. But as common for formal methods and software engineering practice, a trade off between expressiveness and efficient testable system properties exists.

A *compositional* language and building *modular* software systems has been identified as essential for successful designing. But the classical Petri net formalism as well as first approaches towards higher-level concepts [20, 31] fail to provide it. Still most extensions put their emphasis on preserving an analyzable model while in practice a clear semantics as well as support for embedding based on abstractions like interfaces are more important. *Meyer* [37, p. 979] summarizes the common critiques as follows:

> Petri nets, in particular, rely on graphical descriptions of the transitions. Although intuitive for simple hardware devices, such techniques quickly yield a combinatorial explosion in the number of states and transitions, and make it hard to work hierarchically (specifying subsystems independently, then recursively embedding their specifications in those of bigger systems). So they do not seem applicable to large, evolutionary software systems.

In general is behavior modeling neglected in practice while several newer trend like the shift towards *software architecture* [46] may change this. Nowadays, software evolves from isolated solutions for business or industry applications towards distributed environments. It will further interlink information system structures which are still isolated today and become *persuasive*. The software will often take responsibility for considerable coordination tasks and the ever increasing demand to improve business processes and process centered technologies like *workflow* [45] indicates that this trend will make system behavior one essential point.

Formal methods and from our point of view especially Petri nets can gain more acceptance from the resulting change of demands. This is independent from the fact whether this trend leads to a common software engineering practice where complete analyzable system models will become the regular case. The trend towards higher-level abstraction to handle the ever increasing *complexity* has led to the success of visual notations in software engineering for structure modeling. For behavior modeling to achieve more acceptance also a notation which is *scalable* as well as has an intuitive semantics is needed. The notation has to cover concurrency as a specific aspect, design has to deal with in the future. Petri nets conceptionally provide ingredients for all these aspects.

The *object coordination net* (OCoN) approach [57, 24, 27] tries to overcome the described problems with high-level Petri net formalisms. It has its origin and roots in an attempt to achieve an equally weighted compromise between the requirements of concurrency modeling (due to the usage of Petri nets), object-orientation for structure and behavior and the limits and demands a suitable visual formalism implies.

In the paper the stepwise development of the OCoN approach foundations is presented. In section 2 the basic notion of contract and its formalization in terms of protocol nets is studied. Then, a flexible notion of port-passing nets named *coordination nets* (CoN) is introduced in section 3. It is used to define the OCoN notation in section 4 on top of them in combination with the UML. Visual language aspects are discussed in section 5 and the tight integration with the UML is studied in section 6 by presenting a schematic example. We finally compare the approach with the most prominent proposed solutions for object-oriented nets in section 7 and discuss other related work. We conclude the article with some remarks on planned further work and the project status.

## 2 Protocol Nets and Contracts

When behavioral modeling should also provide some degree of modularity, abstraction and data hiding [41] are mandatory. While several extensions to the classical interface notion to achieve a more suitable external specification have been suggested, is the question of behavior w.r.t. subtyping and inheritance still an area of research. The phenomena of *non uniform service availability* for a class has to be considered for interfaces to provide the needed encapsulation and separation. For the *object-oriented design* statecharts [28] for OMT [44] or *path expressions* for Fusion[15] have been proposed. Both are used to specify the external available operations of a class depending on its history.

A general notion of a *contract* covering the classical as well as additional aspects is presented in [6]: The cases of *syntactical interfaces*, *behavior contracts*, *synchronization* effects and *quality of service* are distinguished. While *syntactical interfaces* do not provide enough information to exclude semantical *misusage*, can *behavior contracts* not be managed in an efficient way automatically. Quality of service considerations are often run-time dependent and thus can only be specified when instantiating a system on a specific platform. In contrast does the *synchronization* aspect represent an aspect that can be considered already during the design and can be expressed using Petri nets.

The notion of *substitutability* [55] can be used to characterize *behavior subtyping* [1] needed to ensure the secure usage of contracts w.r.t. behavior in contrast to *interface subtyping* as supported by most object-oriented languages. When multiple concurrent clients are possible we also have to ensure *view consistency* [35] for a subtype.

### Protocol Nets

The identified demand of covering synchronization aspects within contracts is handled in the OCoN approach by supporting the specification of protocols with a (nearly) state machine like labeled place/transition net. In order to distinguish if a behavior has to occur (obligation) or may be used as needed we have to further distinguish between *fair* transitions and *quiescent* (grey) ones that do guarantee some sort of *progress* or not (cf. [43]).

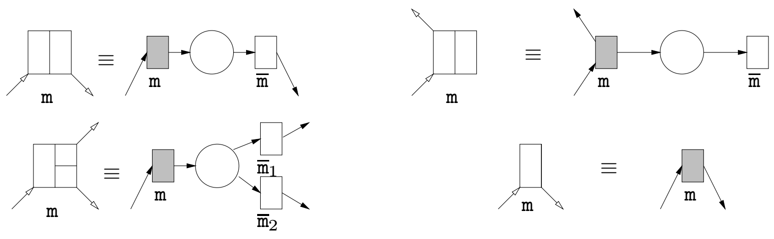


**Fig. 1.** The set of protocol macros

In figure 1 the different kind of operations which are supported by an OCoN protocol net are defined. The protocols are specified from the perspective of the client and thus a grey behavior indicates free choice while a normal transition has to occur. The normal labels indicate a usual or one way request (`m`) while a label $\overline{\mathtt{m}}$ does correspond to a reply or event. Only the modification and synchronization w.r.t. the protocol state itself is described in a protocol net and thus edges for parameter or reply values do not occur. From left to right and top to the bottom we have the usual operation request containing of a request as well as a reply, an operation request with parallel reply, a request with alternative replies (e.g., named replies or exceptions can be covered) and an one way call.

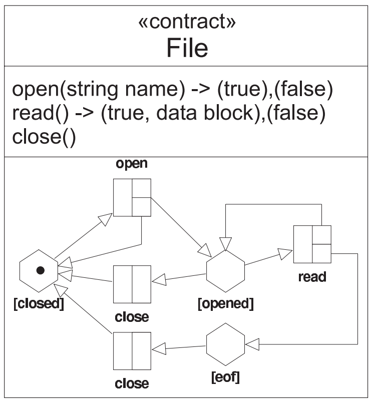

«contract»
File
open(string name) -> (true),(false)
read() -> (true, data block),(false)
close()
open
read
[closed]
close
[opened]
close
[eof]

**Fig. 2.** A File contract described with a protocol net

In practice, for example, a file handle protocol will imply a certain usage, e.g., a read request will not succeed if not an open request has succeeded before or if the end of file is already reached. This example protocol of a File can be described using the defined macros of figure 1. An appropriate protocol for a file handle with operations `open`, `read` and `close` is presented in figure 2. The initial state where only an `open` request is possible is named [closed]. One possible reply is $\overline{\mathtt{open}}_1$ as an acknowledgment for a successful opening which results in state [opened]. If the request fails, an *exception* $\overline{\mathtt{open}}_e$ is replied and the protocol remains in state [closed]. A file handle in state [opened] can further be used to read data. A successful `read` request is signaled by reply $\overline{\texttt{read}}_1$ whereas the reached end of file results in $\overline{\texttt{read}}_2$ and a state change to [eof]. If we are either in state [opened] or [eof] the `close` request can be used to close the file handle and reach the state [closed] again.

The contracts are the essential elements to separate the classes as well as allow further independent subsystem evolution. Subtyping and contract inheritance are suitable concepts to support such efforts. For the OCoN approach a contract subtyping notion has been developed in [23] that provides the needed *substitutability* as well as *view consistency*.

While the described protocol nets are capable to describe the behavior related to a single connection in form of a protocol, we have to provide a formalism that is also capable of expressing multiple connections in parallel.

## 3 Coordination Nets

For *colored Petri nets* the extension to *hierarchical colored Petri nets* [32] and the composition mechanisms *substitution of transitions or places*, *invocation transitions* and *fusion sets for transitions or places* have been proposed [30]. All these mechanisms, excluding the invocation, are very Petri net specific and do not rely on the natural notion of information exchange directly, but encode it into a net specific view. Consider for example a *place fusion* which might be a useful abstraction in a assembly line like structure, but it does not correspond to a common software interface like a procedure or stream. Nets with procedure calls as considered in [33] result in considerable analysis problems and thus are usually provided as add-on and not as basic concept; see,e.g., [19] for an extension of B(PN)$^2$ [5] with procedures. The *transition invocation* can be considered as the procedural abstraction common in programming languages and thus provides the needed general abstraction concept. To model object orientation and dynamically evolving structures we even have to add references and *port passing* capabilities (cf. $\pi$-calculus [38]).

The *coordination nets* (CoN) formalism has been invented in [21] to provide an abstraction for the specification of the OCoN semantics. We base the formalization on the forthcoming ISO high-level Petri net standard [16], which provides a non hierarchical high-level Petri net model. Port passing capabilities to model instance and system behavior even for dynamic evolving structures are added. For the CoN formalism, the level 2 conformance with the *HLPN* standard has been demonstrated in [21]. In extension to high-level Petri nets as defined in the standard a concept to provide *modularity* is needed. To achieve this, a system is built based on a set of *coordination net graphs* which are allowed to interact. The formalism should allow to model multiple instances of one object type, each one providing a set of interfaces with dynamically changing external protocol state. Several net instances may be used to implement the object behavior together and thus a mechanism like place sharing for them is necessary to model the object environment shared by all net instances of an object instance.

We do not provide a builtin object notion with the *coordination nets*. Instead, we provide an interface based separation using typed ports which consist of an interface and a protocol net restricting the message occurrences. Our final net dialect *object coordination nets* will provide a suitable object and class notion based on CoNs.

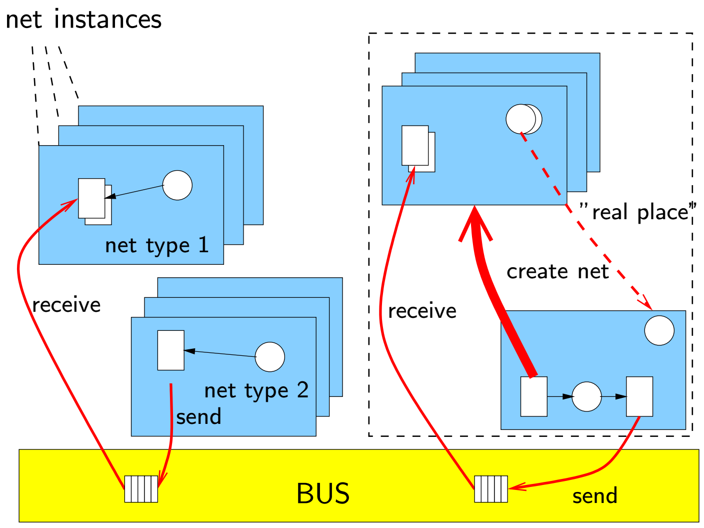


**Fig. 3.** Basic structure of a *coordination net system*

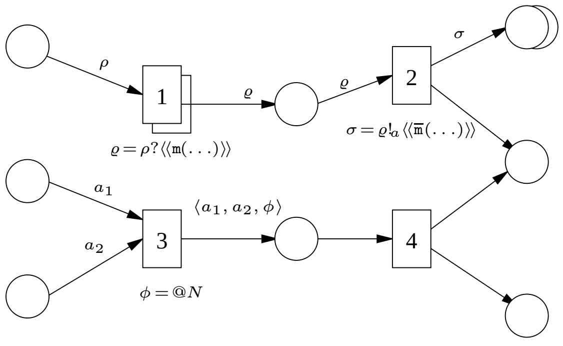
The standard is extended by suitable mechanisms for communication, place sharing as well as dynamic net and port creation. In figure 3, the basic structure of a *coordination net system* is shown. There do exist multiple instances of the same net type possibly sharing places between a net instance and its child ("real place arrow connection"). The nets may communicate using a communication *infrastructure*. The basic idea for communication is to introduce ports $\zeta$, $\eta$, ... representing associated or exported interfaces (objects) as pairs of connected communication endpoints which are represented by port token (see figure 4, 5 for port usage). These ports can be used to receive a message (?) using the following annotation for a transition:

$$\eta = \zeta?\langle\!\langle \mathsf{op}(a_1, \ldots, a_n)\rangle\!\rangle \qquad \eta = \zeta!_s\langle\!\langle \mathsf{op}(\ldots)\rangle\!\rangle \qquad \eta = \zeta!_a\langle\!\langle \mathsf{op}(\ldots)\rangle\!\rangle,$$

where $\eta$ is the resulting port and $\langle\!\langle\rangle\!\rangle$ denotes a given *marshaling function*; $\mathsf{op}(a_1, \ldots, a_n)$ stands for an operation call with operation name $\mathsf{op}$ and input parameters $a_1$, ..., $a_n$. There may be several distinct return vectors for a call and thus we use $\overline{\mathsf{op}}_i$ as operation name for the return alternative $i$ to an operation $\mathsf{op}$ and annotate $\overline{\mathsf{op}}_i(r_1, \ldots, r_m)$ for a reply with return vector $r_1$, ..., $r_m$. A corresponding *synchronous* send can be specified using a port $\zeta$ and the synchronous send operator $!_s$. Analogous an *asynchronous* send can be specified using $!_a$ We distinguish *provide ports* $\rho$, $\varrho$, ... for exported and *usage ports* $\phi$,

$\varphi$, ... for associated interfaces. A provide port can receive operation calls `op` and sends replies $\overline{\texttt{op}}_i$ whereas an usage port can be used to send requests `op` and receive replies $\overline{\texttt{op}}_i$. Asynchronous and synchronous interaction are distinguished, because the synchronous interaction provides more sophisticated ways to interact. The asynchronous communication is in contrast more efficient and reduces the coupling between two systems. When useful we do not further specify if synchronous or asynchronous interaction is wanted and the more general send operation (!) is used.

To create ports of type $P$ or net instances for a declared net type $N$ also corresponding annotation expressions are supported. A net creation expression ($\phi = @N$) binds to $\phi$ an usage port corresponding to a special initial provided `st`andard port (`std`) each new net instance contains. This initial connection allows to establish more connections by using these port connections to publish other ones. After a port creation ($(\rho,\phi) = @P$) a pair of new unique connected usage and provide port instances is bound to $\phi$ and $\rho$. This way the dynamic creation of active net instances as basic formalism to model instances as well as multiple active threads of an instance is provided.

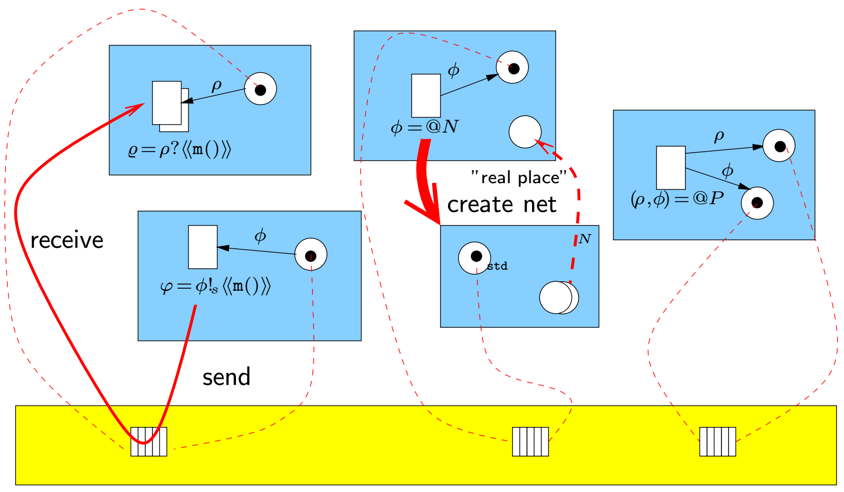


$\rho$
1
$\varrho$
$\varrho$
2
$\sigma$
$\varrho = \rho?\langle\langle \texttt{m}(\ldots)\rangle\rangle$
$\sigma = \varrho!_a\langle\langle \overline{\texttt{m}}(\ldots)\rangle\rangle$
$a_1$
$a_2$
$\langle a_1, a_2, \phi\rangle$
3
4
$\phi = @N$

**Fig. 4.** A *coordination net graph* example

By allowing the described annotations in net declarations, a dynamically changing set of net instances interacting via port instances can be specified. To achieve a better visual representation we draw all transitions annotated with receive terms and imported places with a shadow as shown in figure 4. There is a request received in transition 1 which is replied with a send expression in transition 2. Transition 3 creates a new net of type $N$ and propagates the resulting usage port $\phi$ together with its other pre-conditions $a_1$ and $a_2$ in form of a vector to a place. Transition 4 may consume it then. Hence, the parts which a single coordination net graph distinguishes from a high-level Petri net as defined in the high-level Petri net standard are the additional annotations. They add message send, message receive, port creation and net creation to a transition as shown in figure 4.

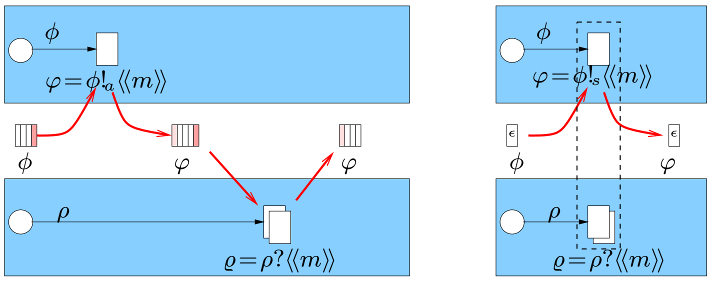
The single net graphs are interacting via send and receive annotations which are using the already introduced ports as addresses. The resulting system consists



$\rho$
$\varrho = \rho?\langle\langle \mathtt{m}() \rangle\rangle$
$\phi$
$\phi = @N$
"real place"
create net
$N$
std
$\rho$
$\phi$
$(\rho,\phi) = @P$
receive
$\phi$
$\varphi = \phi!_s\langle\langle \mathtt{m}() \rangle\rangle$
send

**Fig. 5.** An example for different ways nets may interact

of a number of net graph instances connected via ports and a marking for all of them, as presented in figure 5. The left two nets interact via corresponding usage/provide ports and a synchronized send and receive transition pair. The *synchronous send* ensures, that the message is directly received. In the middle a net creation is presented and the resulting port pair is visualized. The imported place of the created Net $N$ is linked to the corresponding local place of the creating net and the standard port (`std`) of the new net is connected to the resulting port $\phi$ of the create expression. A port creation is demonstrated in the right net. This technique is used to describe instance or subsystem wide sharing of resources and is realized with some sort of *lexical scoping*.

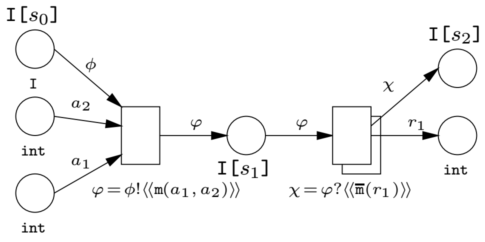


$\phi$
$\varphi = \phi!_a\langle\langle m \rangle\rangle$
$\phi$
$\varphi$
$\varphi$
$\rho$
$\varrho = \rho?\langle\langle m \rangle\rangle$
$\phi$
$\varphi = \phi!_s\langle\langle m \rangle\rangle$
$\epsilon$
$\epsilon$
$\phi$
$\varphi$
$\rho$
$\varrho = \rho?\langle\langle m \rangle\rangle$

**Fig. 6.** Asynchronous and synchronous interaction

We have decided to provide synchronous as well as asynchronous behavior for coordination nets to achieve a greater flexibility. The *synchronous* interaction (cf. *synchronous channels* [14]) is useful, because it provides a higher-level abstraction to describe explicit synchronization where needed while the asynchronous interaction can be used to combine systems with FIFO queues (cf. *FIFO Petri* *nets* as a medium [48]). An example visualization of an asynchronous and synchronous interaction is shown in figure 6. In the left case of an asynchronous interaction the FIFO queue as medium does decouple both transitions while in the right case of synchronous interaction both transitions are firing atomically together.

A suitable typing has to carefully distinguish usual types (literals) that describe a value domain and represent *passive* data and *ports* which are handles or identifiers that allow to request certain activities or attributes. For ports a typing that supports *subtype polymorphism* is essential. Ports represent connections to other entities in a fashion that should ensure *abstraction* and *autonomy* which are the essential characteristics of *object-based* systems (see *Wegner* [54]). The basic idea for *port typing* is to associate an *interface* (*signature*) and a *behavior* to every port connection. Thus the *object life cycle* and the possible *interaction* with an object becomes a part of the usage contract.



I[$s_0$]
I
$\phi$
$a_2$
int
$a_1$
int
$\varphi$
I[$s_1$]
$\varphi$
$\varphi = \phi!\langle\langle$m$(a_1, a_2)\rangle\rangle$
$\chi = \varphi?\langle\langle\overline{\text{m}}(r_1)\rangle\rangle$
$\chi$
I[$s_2$]
$r_1$
int

**Fig. 7.** The usage side of a *remote procedure call*

A protocol conform usage is presented in figure 7 where the port $\phi$ is transformed to $\varphi$ by sending m and later transformed to $\chi$ when receiving the reply $\overline{\text{m}}$. The port type and state is annotated using the shortcut I[$s_i$], where I denotes the *interface* and *protocol* and $s_i$ the specific protocol state.

## 4 Object Coordination Nets

The *object coordination nets* (OCoNs) combine the strength of Petri nets with the structural modeling techniques of the UML, the de-facto standard for object-oriented modeling. By combining both techniques in an orthogonal way, the Petri net mechanisms can be used to express coordination, concurrency and partial states while the structure is described in terms of objects, classes and associations. The Petri net concepts lack a suitable structural modeling concept and thus an orthogonal combination with the object-oriented structural model is needed and possible. OCoNs are build on top of the CoN formalism to provide a more high-level as well as more restricted formalism. Structural aspects are realized with UML mechanisms and the usage of nets is restricted to model behavior. As demonstrated in [25], the OCoN formalism results in a more accurate representation of the structural model within the formalism compared with statecharts.

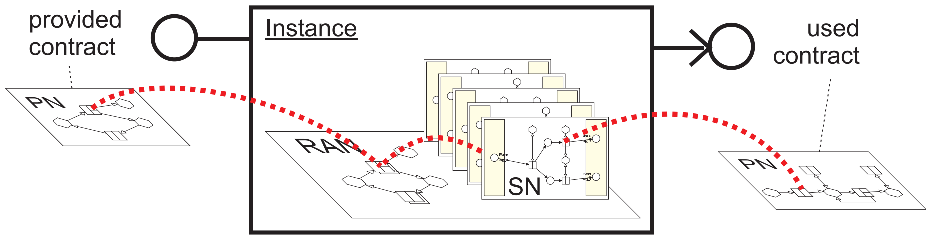


**Fig. 8.** The structural concept for a class or subsystem

The CoN formalism is used to describe two specific net forms while the protocol nets are used to type contracts. Figure 8 presents the relation between the different nets. The protocol nets are used to describe the guaranteed or assumed behavior of contracts. So called service nets (SN) describe behavior for a specific task right like a method with its own thread of control in a programming language. An instance wide unique resource allocation net (RAN) is used to describe the overall instance activities like request acceptance as well as the allocation of needed resources for the request processing including the creation of related service nets for a request.

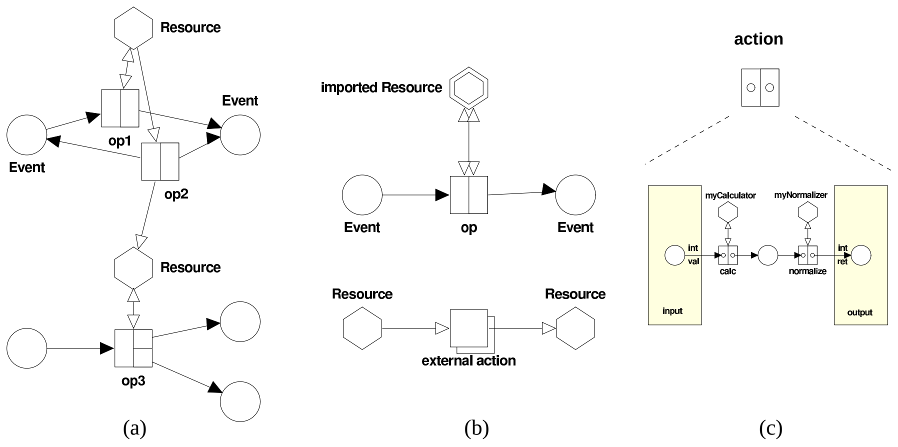


**Fig. 9.** Several basic elements of an OCoN

An overview about the elements of an OCoN is presented in figure 9. The *resource* and *event pool* elements are represented by hexagons and cycles. We distinguish between them, to describe the more transient character of parameters, the control flow and temporary events by using event pools as well as the more static resource character of associations and local variables represented by resource pools. Resources are required as the carriers of activities to perform the processing of events during the computation and thus events describe the control flow of a net (see figure 9 (a) and (b)). Based on the distinction between resources and objects produced and consumed through the flow of data and control, the metaphor of resources which is crucial in distributed systems can be used to make resource handling explicit. The *usage* and *status* of resources can be specified in detail. A single resource may be represented by more than one resource pool if the different pools stand for the same resource but different external states. Using a set of useful actions to abstraction from concrete and error prone explicit port handling, the behavior can be described in terms of requests and simple contract usages. No explicit send and receive have to be considered any more and instead the more high-level interactions like *call* or *one way call* are provided directly. The typing of the contracts using interfaces and protocol nets does further enforce a disciplined usage. Also the creation of instances as well as subnets is covered using extra forms of actions.

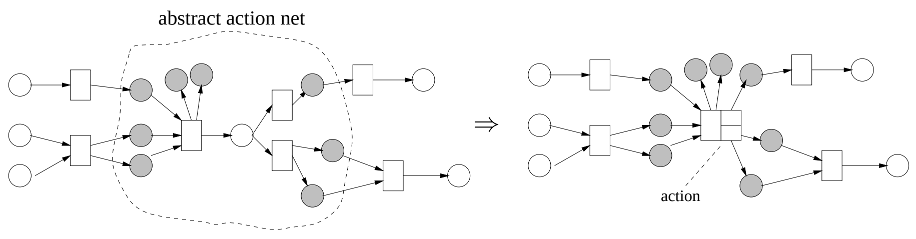


**Fig. 10.** Embedding of an action with multiple output alternatives

To specify the semantics of OCoN constructs we will use *reentrant* subnets in a macro like way and a special kind of *transition refinements* (see [8]). *Valette* [50] refines transitions by subnets called *block* with one initial and one final transition. The block is protected from multiple occurrences of the initial transition before the final transition occurs by assuming that the refined transition is not 2-enabled for any reachable marking. Thus the net must not be *reentrant*. *Suzuki* and *Murata* [49] generalize this technique and consider the case, where the refined transition is restricted to be at most $k$-enabled. Work which considers also distributed input and output has been done by *Vogler* [53]. He studies a refinement notion depending on the environment of the transition. He demonstrates that only non distributed input is feasible but distributed output can be used when environment independent refinement is considered. In contrast to all these considerations we need a really *reentrant* (see [13]) construction, otherwise the parallel occurrence of actions will be limited, but for the presented refinements this condition is obviously fulfilled due to their restriction to at most a single token per usage. The general scheme used is presented in figure 10. This way the call with contract blocking character, the call with parallel reply or a one way call can be specified.

In figure 11 the corresponding CoN behavior for a regular call with alternative replies is specified. It is a generalization of the simple call described earlier in figure 7. The pre-condition edge denotes the necessary port and a request $\mathtt{m}(a_1, \ldots, a_n)$ is send using the usage port $\phi$. For each possible reply immediately a receive is offered which may handle different return parameters as well as the different transitions can be used to specify different side effects in the embedding net.

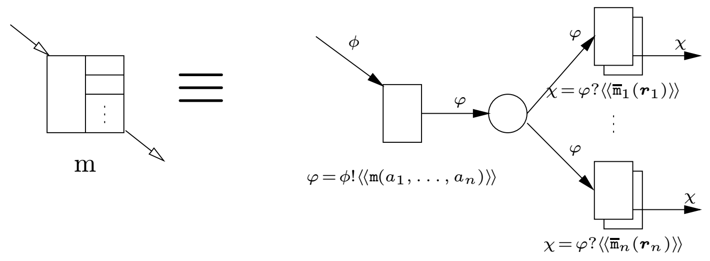

m
$\equiv$
$\phi$
$\varphi$
$\varphi$
$\chi$
$\chi = \varphi?\langle\langle\overline{\mathtt{m}}_1(\boldsymbol{r}_1)\rangle\rangle$
$\varphi$
$\varphi = \phi!\langle\langle\mathtt{m}(a_1, \ldots, a_n)\rangle\rangle$
$\chi$
$\chi = \varphi?\langle\langle\overline{\mathtt{m}}_n(\boldsymbol{r}_n)\rangle\rangle$

**Fig. 11.** The macro refinement for a regular call

As reverse representation exists for every contract usage port also a provide port and the owning instance has to provide the described services accordingly to the protocol net. This has to be done using a so called call forward action as specified in figure 12. Initially a request is received and the set of resources exclusivly needed for the request processing is consumed. As post-condition the received parameters as well as the allocated resource are forwarded to a new created net. Also the port for receiving the result from the new instantiated net as well as the port to send the reply to the requesting party are stored as a pair locally. The created service net will initially receive the forwarded request. When it terminates it will send the reply and the temporary used resources back. The reply will be forwarded while the resources are put back to their original pools. A call action may occur in a service or resource allocation net while a call forward action for the request acceptance is restricted to occur only in a resource allocation net.

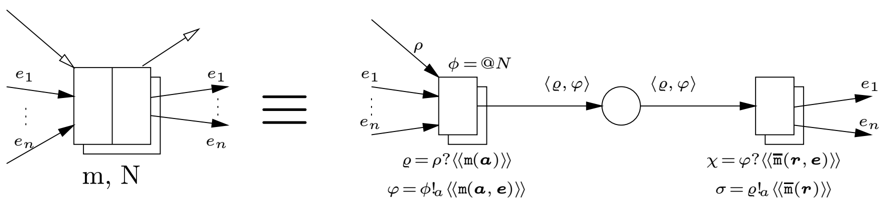

$e_1$
$e_n$
$e_1$
$e_n$
m, N
$\equiv$
$\rho$
$\phi = @N$
$e_1$
$e_n$
$\langle\varrho, \varphi\rangle$
$\langle\varrho, \varphi\rangle$
$e_1$
$e_n$
$\varrho = \rho?\langle\langle\mathtt{m}(\boldsymbol{a})\rangle\rangle$
$\varphi = \phi!_a\langle\langle\mathtt{m}(\boldsymbol{a}, \boldsymbol{e})\rangle\rangle$
$\chi = \varphi?\langle\langle\overline{\mathtt{m}}(\boldsymbol{r}, \boldsymbol{e})\rangle\rangle$
$\sigma = \varrho!_a\langle\langle\overline{\mathtt{m}}(\boldsymbol{r})\rangle\rangle$

**Fig. 12.** The call forward action and request acceptance

When considering the definitions for a call action of figure 11 directly on the OCoN net level, we can see that a labeled action does essentially fire two times during the processing of a request. One time when the request is started and once when it terminates. This step semantic is further described in figure 13. The two steps correspond very well to the input and output parts of the call action. While the direct correspondence with a classical Petri net transition is

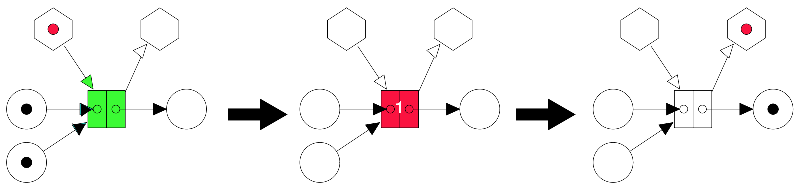


**Fig. 13.** The two times an action can fire

abandoned, the integrity of an operation request including request sending and reply receiving is better preserved this way.

By additionally providing a notion of *inheritance*, the OCoN language can be considered to fulfill the requirements for an *object-oriented* language (cf. [54]). The notion of *inheritance* for concurrent object oriented languages is a critical design aspect. As noted already in [1], inheritance can be employed to reuse the sequential methods, but inheriting the instance wide synchronization seems to be not practical. Thus in the current used inheritance notion for OCoN classes, a subclass does inherit all structural properties as well as the associated service nets of its superclasses, while the resourc allocation net is not inherited. Due to the syntactical inheritance it is ensured that each subclass contains always all resources a supertype service net may demand. The overall resource allocation of a derived class has to be rewritten and reuse is currently not supported for resource allocation nets.

We have to emphasize that in contrast to the contract subtyping and inheritence, *implementation inheritance* is for the intended usage of OCoNs not that relevant. The external visible contract or interface hierarchy should be in general better separated from implementation reuse strategies applying inheritance, otherwise later when the subsystems evolve independently serious design problems will result.

## 5 Visual Language

For place/transition nets the popular *token game* provides a suitable visual representation as well as intuitive semantics. We thus have designed the OCoN formalism to provide a set of higher-level action types that can be understood w.r.t. the local effects as a token game. E.g., the enabling does not depend on textual guards. To model alternative behavior in a graphical rather than textual manner we use methods or external operations with alternative replies (see figure 11) that indicate the relevant different cases. This way a useful additional abstraction for predicates is introduced and textual guards transferring the semantics from the transitions to the annotations can be avoided. This is in contrast to most *HLPN* approaches which make heavy use of textual annotations and are thus not such suitable visual formalisms.

In figure 14 the visual integration of contracts described by protocol nets and the resource allocation as well as service nets is presented. Associations repre-

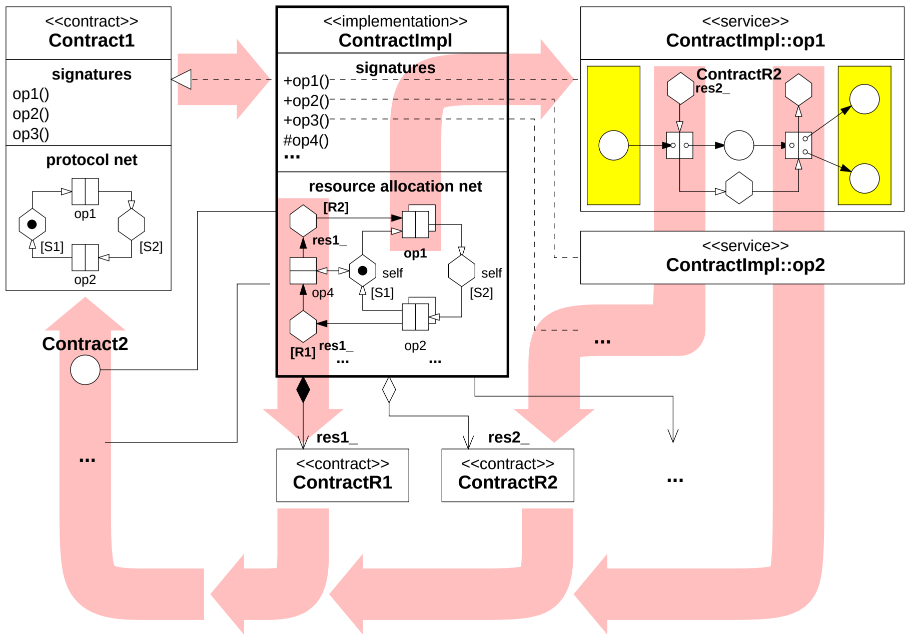


**Fig. 14.** Visual *seamless* embedding of contracts via typed resource pools

sented by resource pools containing related contracts can be used in conformance with the specified protocol and thus their usage corresponds to a graphical embedding. This *seamless* visual embedding [26] is the reason for our design decision to restrict protocol nets to state machines. Then the multiple places of a Petri net allow to embed the contracts using resource pools representing the different contract states. The object *life-cycle* can also be modeled with full Petri nets, e.g., with subtyping based on branching-bisimulation and abstraction [51], but then the intended visual embedding as well as a more Petri net independent contract notion are excluded. In figure 9 (c) the related signature abstraction relating an action to a servie net has been demonstrated.

## 6 Integration into the UML

In the OCoN approach the *HLPN* standard [16] and the UML [40] have been combined. But in practice usually perfect *orthogonality* is not achieved. The UML specification does still contain several inconsistencies, but there are currently attempts like the pUML initiative [18] that try to improve the situation and thus using it is still more appropriate than chosing a proprietary solution. For the included behavioral description techniques we even identified several weaknesses [22]. For the achieved OCoN integration it has been demonstrated in [25] that several limitations and problems related to behavior modeling with the UML notations can be avoided.

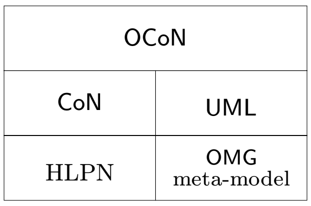


**Fig. 15.** The layers to build the OCoN semantics

In figure 15 the abstraction layers of the semantic foundations of the OCoN approach are visualized. We decided to avoid the considerable weaknesses of the UML by integrating our approach only with a w.r.t. structural as well as behavioral questions more consistent subset.

By providing the bus like implementation structure shown in figure 8 for a class, a clear separation between request specific and overall instance behavior is achieved. In figure 16 an example of a simple complete UML structure with added nets is presented. It describes a class SiteImpl that implements a simple allocation protocol which provides alternating allocate and release operation calls. The implementation stores the provided Data in a buffer initially filled with an empty data item. The buffer is accessed using the Buffer contract which is implemented by the class BufferImpl. The three specific stereotypes ≪contract≫ for contracts, ≪implementation≫ for the overall instance behavior including the resource allocation net and ≪service≫ for a service net or method are

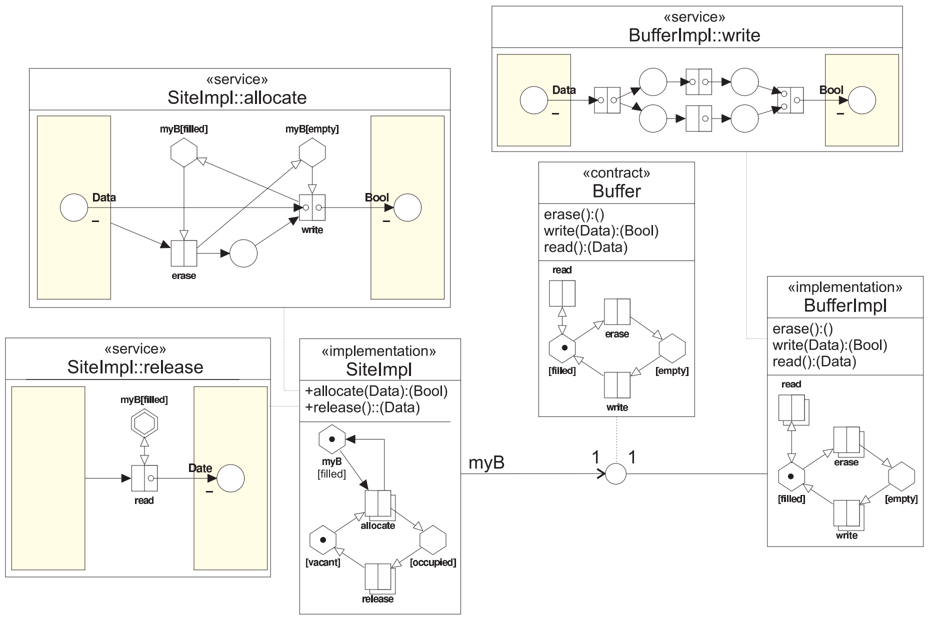


**Fig. 16.** An OCoN and UML example

used. For the operations SiteImpl::release and SiteImpl::allocate the difference between initial exclusive locking and shared access can be described. While for the SiteImpl::release call forward action no initial exclusive locking is specified will SiteImpl::allocate demand it. In correspondence is in SiteImpl::release the considered resource myB shown with a double bordered hexagon to indicate that it is an imported resource and thus interference is possible. In contrast has SiteImpl::allocate an exclusive resource for myB and thus the resource is drawn with a single border. Also the number of used contracts Buffer for SiteImpl are specified in the UML diagram and we can derive the related net capacities using the specified *multiplicity constraints* of the related association.

## 7 Related Work

The net dynamics of the OCoN approach has been realized introducing the visual as well as high-level Petri net conform CoN formalism. In the context of the $\pi$-calculus also a net based calculus named *Mobile Nets* has been developed [11]. It extends the $\pi$-calculus to contain *true concurrency*, but this is done by extending the usual textual binding for $\pi$-calculus processes to cover some notion for places that can be accessed in parallel and thus does not provide the needed visual net related methapor like the CoN formalism neccessary to build a visual language upon.

We think it is more promising for the system design with Petri nets to avoid a Petri net specific mechanism and integrate Petri nets into an usual object-oriented decomposed system. The approach should rely on the well studied and successful mechanisms for abstraction and encapsulation. Following [3] we can classify most earlier proposed solutions to combine object orientation and Petri nets as either "Petri nets inside Objects" [10] or "Objects inside Petri Nets" [4, 52]. Later approaches support more dynamic and expressive models where object references are controlled in nets related to classes and thus they can support both concepts.

Another cruicial aspect for the design of an object-oriented Petri net formalism is the interaction and if it supports interfaces and polymorphism in a manner adequate for software. Several approaches support the traditional approach to connect Petri nets using *place fusion* [2, 34], but places usually provide no suitable interface directly. Most solutions derived from the algebraic specification domain instead provide cooperation in terms of *transition fusion* [4, 10, 7, 17] and allow the related behavior to access all cooperating objects of that activity in a united action. This results in a missing encapsulation and behavior will not be associated with objects itself which is a common criteria for object-orientation. The support for message exchange or operations is essentially needed to achieve encapsulation. The different approaches that support this vary w.r.t. the level of support for either message passing or the higher-level interaction of a procedure call construct [47, 36, 12, 29]. The approaches provide an object state either explicit using one global net per instance [39, 34, 47, 29] or only implicit as composition of so called method nets [36, 12]. A systematic separation into a resource oriented scheduler describing the overall instance state and method related active method net instances is only realized in the OCoN approach. To achieve visual *scalability* the usage of several nets for specific tasks is necessary while their simple visual separation using regions (cf. [12]) is not sufiicient.

For distributed system design the encapsulation has to be ensured and hence a notion of contract or interfaces is necessary and thus not type secure approaches like [12] are not sufficient. Up to our knowledge does no other approach intergrate an external behavioral specification like a protocol net to provide a contract notion with behavioral subtyping and thus supports behavioral abstraction.

## 8 Conclusion and Outlook

The integration of software engineering and especially object-oriented technology with a high-level Petri net formalism that is extended in a $\pi$-calculus style to also cover dynamic aspects has been presented. The OCoN approach builds an orthogonal extension to a subset of the UML and adds powerful concurrency and contract modeling capabilities. A tight integration has been achieved while proprietary extensions to the UML itself could be avoided.

The contract notion for the OCoN design approach supports the explicit specification of contractually relations and provides a notation to specify coordination aspects already on an abstract level. We can further express several design alternatives and evaluate them [27] in order to decide which one is most suitable. Thus, the OCoN formalism is a suitable notation that can be applied already during the earlier stages of the design process with emphasis on the *software architecture* [46]. A suitable visual notation is a crucial prequisite for a successful approach. We applied useful Petri net visualization concepts and achieved to preserve them by applying object-oriented standard techniques in a systematic fashion.

Our initial application domain has been *distributed software systems*, while we have also explored *embedded systems* and currently investigate the design of *workflow* [56] applications. The experience with student classes and courses indicates that even without experienced designers the approach is applicable. While the results are promising the training for a specific net based notation is still difficult for beginners. A framework supporting the final implementation of OCoN designs as well as an integration of the tools into an UML tool and extensions towards consistency checks and complete simulation capabilities are planned. At the moment, we study the approach also in an industrial environment to gain more experience with larger projects.

## References

1. P. America. A Behavioural Approach to Subtyping in Object-Oriented Languages. Techreport, Philips Research Laboratories, 1989. Technical Report 443.
2. M. Baldassari and G. Bruno. An Environement of Object-Oriented Conceptual Programming Based on PROT Nets. In *Advances in Petri Nets*, number 340 in LNCS, pages 1–19. Springer Verlag, 1988.

3. R. Bastide. Approaches in unifying Petri nets and the Object-Oriented Approach. In *1st Workshop on Object-Oriented Programming and Models of Concurrency, within the 16th International Conference on Application and Theory of Petri nets, 27 June 1995, Turin, Italy*, 1995.
4. E. Battiston, F. D. Cindio, and G. Mauri. Objsa Nets: A Class of High-Level Nets Having Objects as Domains. In *Advances in Petri Nets*, number 424 in LNCS, pages 20–43. Springer Verlag, 1988.
5. E. Best and R. P. Hopkins. B(pn)$^2$ - a Basic Petri Net Programming Notation. In *PARLE'93*, LNCS, pages 379–390. Springer Verlag, 1993.
6. A. Beugnard, J.-M. Jezequel, and D. Watkins. Making Components Contract Aware. *IEEE Computer*, 32(7):38–45, July 1999.
7. O. Biberstein and D. Buchs. Structured Algrbraic Nets with Object-Orientation. In *Applications and Theory of Petri Nets 1995, 16th International Conference, Turin, Italy*, number 935 in LNCS. Springer Verlag, June 1995.
8. W. Brauer, R. Gold, and W. Vogler. A Survey of Behaviour and Equivalence Preserving Refinements of Petri Nets. In *Advances in Petri Nets*, number 483 in LNCS, pages 1–46. Springer Verlag, 1990.
9. W. Brauer, W. Reisig, and G. Rozenberg [eds]. *Petri Nets: Central Models (part I)/Applications (part II)*, volume 254/255 of *LNCS*. Springer Verlag, Berlin, 1987.
10. D. Buchs and N. Guelfi. A Concurrent Object-Oriented Petri Net Approach. In *Applications and Theory of Petri Nets 1991, 12th International Conference, Gjern, Denmark*, 1991.
11. N. Busi. Mobile Petri Nets. In *Proc. 3rd Int. Conf. on Formal Methods for Open Object-based Distributed Systems (FMOODS), February 15-18, 1999, Florence, Italy*, pages 51–66. Kluewer Academic Publishers, 1999.
12. M. Ceska, V. Janousek, and T. Vojnar. PNtalk - A Computerized Tool for Object Oriented Petri Nets Modeling. In *EUROCAST'97, Las Palmas de Gran Canaria, Canary Islands, Spain*, number 1333 in LNCS. Springer Verlag, 1997.
13. G. Chehaibar. Use of Reentrant Nets in Modular Analysis of Colored Nets. volume 524, pages 58–77, Berlin, Germany, 1991. Springer Verlag. NewsletterInfo: 40.
14. S. Christensen and N. D. Hansen. Coloured Petri Nets Extended with Channels for Synchronous Communication. In *LNCS; Application and Theory of Petri Nets 1994, Proceedings 15th International Conference, Zaragoza, Spain*, volume 815, pages 159–178. Springer Verlag, 1994.
15. D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development: The Fusion Method.* Prentice-Hall, 1994.
16. Committee Draft ISO/IEC 15909. *High-level Petri Nets - Concepts, Definitions and Graphical Notation*, Oct. 1997. Version 3.4.
17. J. Engelfriet, G. Leih, and G. Rozenberg. Net-Based Description of Parallel Object-Based Systems. In *Foundations of Object-Oriented Languages*, number 489 in LNCS. Springer Verlag, 1990.
18. A. Evans, R. France, K. Lano, and B. Rumpe. Developing the UMl as a Formal Modelling Notation. In *UML'98 Beyond the notation. International Workshop Mulhouse France.* Ecole Superieure Mulhouse, Universite de Haute-Alsace, 1998.
19. H. Fleischhack and B. Grahlmann. A Petri Net Semantics for B(PN)$^2$ with Procedures. In *Proceedings of PDSE'97 (Parallel and Distributed Software Engineering), Boston MA*, pages 15 – 27. IEEE Computer Society, May 1997.
20. H. J. Genrich and K. Lautenbach. System Modelling with High-Level Petri Nets. *Theor. Comp. Science*, 13:109 – 136, Jan 1981.
21. H. Giese. Object Coordination Nets 2.0 – Semantics Specification. Techreport, University Münster, Computer Science, May 1999. 15/99-I.

22. H. Giese. Towards a Dynamic Model for the UML. In *14th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications November 1-5, 1999, Denver, Colorado, USA. Workshop: Rigorous Modeling and Analysis with the UML: Challenges and Limitations*, Nov. 1999. (submitted statement).
23. H. Giese. Synchronization Behavior Typing for Contracts in Component-based Systems. Techreport, University Münster, Computer Science, Distributed Systems Group, Feb. 2000. Techreport 03/00-I.
24. H. Giese, J. Graf, and G. Wirtz. Modeling Distributed Software Systems with Object Coordination Nets. pages 107–116, July 1998. Proc. Int. Symposium on Software Engineering for Parallel and Distributed Systems (PDSE'98), Kyoto, Japan.
25. H. Giese, J. Graf, and G. Wirtz. Closing the Gap Between Object-Oriented Modeling of Structure and Behavior. In *UML'99 - The Second International Conference on The Unified Modeling Language Fort Collins, Colorado, USA*, volume 1723 of *LNCS*, pages 534–549, Oct. 1999.
26. H. Giese, J. Graf, and G. Wirtz. Seamless Visual Object-Oriented Behavior Modeling for Distributed Software Systems. In *IEEE Symposium On Visual Languages, Tokyo, Japan*, Sept. 1999.
27. H. Giese and G. Wirtz. Early Evaluation of Design Options for Distributed Systems. In *Int. Symposium on Software Engineering for Parallel and Distributed Systems (PDSE'2000), Limerik, Ireland*. IEEE Press, June 2000.
28. D. Harel. Statecharts: A Visual Formalism for complex systems. *Science of Computer Programming*, 3(8):231–274, 1987.
29. T. Holvoet and P. Verbaeten. PN-TOX: a Paradigm and Development Environment for Object Concurrency Specifications. In *1st Workshop on Object-Oriented Programming Models of Concurrency, Turin*, 1995.
30. P. Huber, K. Jensen, and R. M. Shapiro. Hierarchies in Coloured Petri Nets. In *Advances in Petri Nets*, number 483 in LNCS, pages 313–341. Springer Verlag, 1990.
31. K. Jensen. Coloured Petri Nets. In *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986 Part I*, number 254 in LNCS, pages 248–299. Springer Verlag, 1987.
32. K. Jensen. Coloured Petri Nets: A High Level language for System Design and Analysis. In *Advances in Petri Nets*, number 483 in LNCS, pages 342–416. Springer Verlag, 1990.
33. A. Kiehn. Petri Net Systems and their Closure Properties. In *Advances in Petri Nets 1989*, number 424 in LNCS, pages 306–328. Springer Verlag, 1990.
34. C. Lakos. From Coloured Petri Nets to Object Petri Nets. In *Applications and Theory of Petri Nets 1995, 16th International Conference, Turin, Italy*, number 935 in LNCS. Springer Verlag, June 1995.
35. B. Liskov and J. M. Wing. A New Definition of the Subtype Relation. In *Proceedings of the European Conference on Object-Oriented Programming '93*, volume 707 of *LNCS*, pages 118–141, July 1993.
36. C. Maier and D. Moldt. Object Colored Petri Nets - a Formal Technique for Object Oriented Modelling. Workshop PNSE'97, Petri Nets in System Engineering, Modelling, Verification, and Validation, Hamburg, Germany, Sept. 1997.
37. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997. 2nd edition.
38. R. Milner, J. G. Parrow, and D. J. Walker. A Calculus of Mobiler Processes. Techreport, Edinburgh Univeristy, 1989. Part I and II. ESC-LFCS-89-85/86.

39. A. Newman, S. M. Shatz, and X. Xie. An Approach to Object System Modeling by State-Based Object Petri Nets. *Int. Journal of Circuits, Systems, and Computers*, 9(1):1–20, Feb. 1998.
40. Object Management Group. *OMG Unified Modelling Language 1.3*, June 1999. OMG document ad/99-06-08.
41. D. L. Parnas. A Technique for Software Module Specification with Examples. *Communications of the ACM*, 15(5):330–336, 1972.
42. S. L. Pfleeger. *Software Engineering: Theory and Practice, 1/e.* Prentice Hall, 1998.
43. W. Reisig. Petri Net Models of Distributed Algorithms. In *Computer Science Today – Recent trends and Developments*, number 1000 in LNCS, pages 441–454. Springer Verlag, 1995.
44. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.
45. T. Schael. *Workflow Management Systems for Process Organizations.* Number 1096 in LNCS. Springer Verlag, 1998. Second Edition.
46. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an emerging Discipline.* Prentice Hall, 1996.
47. C. Sibertin-Blanc. Cooperative NETs. In *Applications and Theory of Petri Nets 1994, 15th International Conference, Zaragoza, Spain*, number 815 in LNCS, pages 471–490. Springer Verlag, June 1994.
48. Y. Souissi and G. Memmi. Composition of Nets via a Communication Medium. In *Advances in Petri Nets*, number 483 in LNCS, pages 457–470. Springer Verlag, 1990.
49. I. Suzuki and T. Murata. A method for stepwise refinement and abstraction of Petri nets. *Journal Computer System Science*, 27:51–76, 1983.
50. R. Valette. Analysis of nets by stepwise refinement. *Journal Computer System Science*, 18:35–46, 1979.
51. W. M. P. van der Aalst and T. Basten. Life-Cycle Inheritance: A Petri-Net-Based Approach. In *18th International Conference on Application and Theory of Petri Nets, Toulouse, France, June 1997*, LNCS, pages 62–81, 1997.
52. K. van Hee and P. Verkoulen. Integration of a Data Model and High Level Petri Nets. In *Proceedings of the 12th International Conference on Application and Theory of Petri Nets,Aarhus, Denmark*, pages 410–431, June 1991.
53. W. Vogler. Behaviour preserving refinements of Petri nets. In *Graph-Theoretic Concepts in Computer Science, Proc. WG86, Bernried*, volume 246 of *LNCS*, pages 82–93, 1987.
54. P. Wegner. Dimensions of Object-Based Language Design. In *Object-oriented Programming Systems, Languages and Applications (OOPSLA87, Orlando, Florida, October 4-8, 1987*, volume 22 of *SPECIAl ISSUE of ACM SIGPLAN notices*, pages 168–182. ACM Press, Dec. 1987.
55. P. Wegner and S. B. Zdonik. Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like. In *Proceedings of the European Conference on Object-Oriented Programming '88*, volume 322 of *LNCS*, pages 55–77, Aug. 1988.
56. G. Wirtz and H. Giese. Using UML and Object-Coordination-Nets for workflow specification. In *IEEE International Conference on Systems, Man, and Cybernetics (SMC'2000), Nashville, TN, USA, October 8-11*, 2000.
57. G. Wirtz, J. Graf, and H. Giese. Ruling the Behavior of Distributed Software Components. In *Proc. Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'97), Las Vegas, Nevada*, July 1997.

# Seamless Object-Oriented Software Development on a Formal Base

Stephan Philippi

University of Koblenz-Landau,
Rheinau 1, 56075 Koblenz, Germany
philippi@uni-koblenz.de

**Abstract** Object-oriented development of complex software systems is widely recognized as state of the art within the industry as well as the scientific community. What is less commonly recognized (especially in the industry) is that object-orientation itself is not properly defined and neither are popular notations like UML and others. Existing proposals for the formally based development of object-oriented systems are for different reasons often not usable for complex and/or concurrent systems. In addition, the semantics of object-oriented concepts within such notations only rarely match common programming language implementations. This mismatch most likely leads to a costly redesign of a given model during implementation. To overcome these problems approaches for a seamless object-oriented software development on a formal base are needed.

This article surveys several proposals for the formally based development of object-oriented systems based on Petri-Nets. Subsequently, a new approach in this area is introduced which supports multiperspective modeling of concurrent object-oriented systems on arbitrary abstraction levels and also allows automatic generation of executable Java code.

*Keywords:* concurrent systems design, Petri-Nets, Java code-generation

## 1 Introduction

Today object-oriented software development practice often includes multiperspective system views using different types of diagrams. The most popular collection of such diagrams is the so-called unified modeling language (UML) [Rati99]. Even if the emergence of this industry standard is useful from an economical point of view, there are some serious drawbacks with respect to its application in the software engineering area. Besides the difficulties in choosing an adequate subset of diagrams with respect to a specific application domain, further problems arise from the UML's lack of formality. In fact, most of the UML notations are not formally based, i.e. they have no formally defined semantics. In combination with the fact that a commonly agreed definition of 'object-orientation' does not exist, this easily leads to communication problems, as neither the notations nor the underlying concepts are properly defined. Another problem in this context is that a tight integration of different views in a formal sense is not given with the UML. Thus, contradicting system views representing a non-consistent model are usual observations within UML projects. Especially considering concurrent systems this is not tolerable, as even for small systems a huge amount of possible interaction sequences between concurrent parts exists, which results in difficulties for human understanding. Thus, non-formally based approaches are not well suited to meet today's demands as they do not offer a reliable base for communication and understanding.

Principally, formally based approaches for the object-oriented modeling of systems are well suited to solve these problems. Nevertheless, proposals in this area are not widely accepted today. Major drawbacks of approaches like OOZE [AleGog91], VDM++ [Durr92], and others are that often no visual representation of models exists and the development of concurrent systems as well as simulation is not supported. In addition, such approaches are frequently criticised for being too difficult to use[1]. Another crucial point is that the definitions of object-oriented concepts in formally based notations only rarely match their counterparts in programming languages. Theoretically this is not necessarily a problem, as the realization of object-oriented concepts within common programming languages is not perfect at all, as shows for example the nonvariant instead of covariant overwriting in Java [ArnGos96]. From a more practical point of view the use of such formally based notations leads to a mismatch if a given model serves as architectural layout for the implementation of a system, which is usually the motivation for creating a model in the software engineering area.

Another family of approaches to model object-oriented systems in a formal way is based on Petri-Nets [Petri62], which are well known for their graphical appearance, their simulation capabilities, and their native support for the modeling of concurrent systems. The extension of Petri-Nets with object-oriented concepts is a promising approach as the resulting notation ideally allows for the formally based modeling of concurrent object-oriented systems *and* for the object-oriented structuring of Petri-Nets. Due to this potential, there has been considerable research activity in the object-oriented Petri-Net area.

The next section of this article describes from a software engineering point of view a set of properties for approaches integrating object-oriented concepts and Petri-Nets. Based on these properties, existing proposals are surveyed and typical problems discussed. Section three introduces OOPr/T-Models, a novel approach for the integration of Petri-Nets and object-oriented concepts, which supports the multiperspective modeling of systems on arbitrary abstraction levels. Section four describes the formal base of OOPr/T-Models as well as their automatic translation to executable Java code. Finally, the last section gives a summary and presents further perspectives.

[1] A more detailed discussion of related problems with existing proposals in the area of formally based development of object-oriented systems can be found in [LanHau94]. A survey on UML formalization approaches is given in [Evans*98], [KeEvRu99].

## 2 Essential properties and related work

Since the mid-eighties, various approaches integrating object-oriented concepts and Petri-Nets have been published. To be able to evaluate the existing proposals with respect to their applicability in the software development, a set of essential properties is introduced. The classification of these properties as 'essential' reflects that from our point of view, omitting one of them results in a notation which is not well suited for the object-oriented development of complex, concurrent software systems. Thus, to solve the above stated problems of (non-) formally based notations, an approach integrating object-oriented concepts and Petri-Nets ideally fulfills the following criteria:

- **Completeness with respect to object-orientation**: In order to be able to structure systems in an object-oriented way, a notation combining object-orientation and Petri-Nets should at least support object identity, complex objects, classes, encapsulation, inheritance, overriding, and polymorphism/late binding.

- **Support for seamless development**: Ideally, a notation for the modeling of object-oriented systems supports every stage of development ranging from high-level analysis to low-level implementation. If a notation only supports part(s) of the development cycle, every shift to another notation (e.g. a programming language) will almost inevitably result in a redesign of the model, as no commonly accepted definition of object-orientation exists. The reason for the need to redesign a given model in this context is that different notations integrate different object-oriented concepts, and corresponding concepts often differ significantly from a semantical point of view, as for instance visibility definitions and inheritance.

  > 'When examining object-oriented solutions, you should check that the method and language, as well as the supporting tools, apply to analysis and design as well as implementation and maintenance. The language, in particular, should be a vehicle for thought which will help you through all stages of your work.' [Meyer97]

- **Completeness with respect to Petri-Nets**: Petri-Nets are well-known for their graphical appearance, their power for the modeling of concurrent systems, and their formal base which allows for simulation and analysis. An approach integrating object-oriented concepts and Petri-Nets should preserve these properties.

- **Concepts to resolve inheritance anomalies**: Ideally, a notation for object-oriented software development allows for the modeling of concurrent systems. As a consequence, concepts to resolve inheritance anomalies [MatYon93] should be supported by such a notation. Inheritance anomalies occur in concurrent object-oriented systems if synchronization conditions are integrated into the functional description of a method. Problems arise here, as within every subclass containing additional methods, the inherited synchronization conditions almost inevitably change. As a consequence, inherited methods have to be redefined in subclasses to integrate the modified synchronization conditions, even if there is no need to do so from a functional point of view. Thus, to be able to benefit from inheritance, a notation for the development of concurrent object-oriented systems needs to integrate concepts to avoid the redefinition of methods.

– **Modeling ergonomics/usability**: System modeling usually starts at a high level of abstraction, the main task being to collect knowledge in order to understand the system domain. As this is a very difficult task in its own right, the designer should not be hampered by a modeling framework which does not offer the highest possible ergonomical degree, i.e. the formalism used to support modeling should be as easy as possible to learn and handle. Thus, modeling ergonomics is one of our main concerns even if this property is not exactly quantifiable. In particular, former users of Petri-Nets or object-oriented concepts should with only minor problems be able to use a new approach combining both. Prerequisite is a 'natural' solution allowing each user to feel familiar with the way concepts he already knew are integrated.

The result of the evaluation of existing approaches for the integration of Petri-Nets and object-oriented concepts is that none of them offers an overall satisfactory solution with respect to the described properties. In detail, approaches like PROT-Nets [BruBal86], OBJSA-Nets [BaDeMa88], POT/POP [EnLeRo90], SimCon [HeeVer91], OOCPN [Engl93], PN-TOX [HolVer95], and others are not complete with respect to our understanding of object-orientation, i.e. important concepts are not integrated, like for example inheritance or dynamic binding. Another common problem from the point of view of software development is that approaches like Object/Behaviour Diagrams [KapSch91], Object-Nets [BoNuFe97], OOPN [Stulle97], and others represent objects as Petri-Net structures. Consequently, dynamic object instantiation is not supported in order to avoid dynamic Petri-Net structures. In turn, such proposals only allow for the modeling of systems with a fixed number of objects which have to be identified during system design. To support the development of software systems, such approaches (which are partly intended to be used for the modeling of technical systems) are not well suited, because objects are usually instantiated and destroyed at a high rate during the runtime of such a system.

Other approaches, e.g. PN-TOX [HolVer95] and OCoN [GiGrWi98], are intended for modeling only certain aspects of object-oriented systems, augmenting notations like OMT [Rumb*91] or UML. Thus, only parts of the resulting models have a formally defined semantics. Another point is that approaches like F-Nets [Deck95], OOPN [Stulle97], and others do not provide a single notation covering the life-cycle a project ranging from analysis to implementation. In detail, modeling the functional parts of an object-oriented system is not supported, which leads to the use of other notations whilst shifting from higher to lower abstraction levels. If functional modeling is supported, not necessarily are methods for the partitioning of the functionality of an object. Such a practice does in consequence not take full advantage of the structuring capabilities of object-orientation, e.g. SimCon [HeeVer91] and OPN [Lakos95].

In contrast to the stated problems with object-oriented concepts, most of the considered approaches are complete with respect to Petri-Nets, i.e. only few proposals like OPM [Burk94] lack a formal base.

Considering the development of concurrent systems, none of the evaluated proposals integrates concepts to resolve inheritance anomalies, even if all of them allow for the modeling of concurrent systems, e.g. [CeJaVo97] and [Maier97].

Finally, modeling ergonomics is generally not considered. In combination with the fact that tool support is mostly not given, this leads to proposals which are practically not usable for the development of complex (software) systems[2].

To summarize, on the one hand none of the considered proposals for the integration of Petri-Nets and object-oriented concepts is without weaknesses with respect to the introduced essential properties. From our point of view, existing object-oriented Petri-Net approaches are thus only of limited use for the development of software systems. On the other hand, each of the criteria (except the integration of concepts to resolve inheritance anomalies) is fulfilled by at least one of the considered proposals. As a consequence, the development of a Petri-Net based notation which is practically usable for the seamless, object-oriented development of complex and/or concurrent software systems should be possible at least from a theoretical point of view.

One of the most common problems with the existing work in the area of object-oriented Petri-Nets is that only few proposals are based on one another, i.e. in most cases the experience from existing work is not taken into account for the development of novel approaches. In contrast, common pitfalls of existing proposals had a direct impact on the development of OOPr/T-Models, which are intended to overcome the limitations of existing approaches with respect to the introduced properties.

## 3 Systems modeling with OOPr/T-Models

This section introduces OOPr/T-Models ('object-oriented Predicate/Transition-Models'), which were developed based on the set of essential properties as well as on the evaluation of existing object-oriented Petri-Net approaches.

First of all, the scenario to set up multiperspective system views with OOPr/T-Models is shown. Then the notations used to describe these views are introduced using an object-oriented version of a simple producer/consumer system. Afterwards, an example on how to resolve inheritance anomalies with OOPr/T-Models is given using an extended version of the initial example. The formal base of OOPr/T-Models as well as their automatic translation to Java code are surveyed in section four.

[2] A more complete overview and detailed evaluation of existing object-oriented Petri-Net proposals is given in [Phil99].

### 3.1 The scenario of OOPr/T-Modeling

The scenario providing an overview on how to use OOPr/T-Models is given in figure 1. Starting from a system to model, different views have to be set up, namely a static, a dynamic, and a functional view. The interdependencies of these views result in the following (cyclic) three-step design process, which applies to arbitrary development stages ranging from high-level analysis to low-level implementation. In the latter case OOPr/T-Models can be used as visual programming language.

1. Usually, the starting point of object-oriented modeling is a class diagram structuring the system domain into classes with their respective relationships. As Petri-Nets themselves are not very well suited to model the static aspects of a system, our approach incorporates a subset of UML class diagrams [Rati99] for this purpose.
2. For each class a dynamic model is defined using Petri-Nets. Here, dynamic models integrate conditions for the activation of methods. This allows for the separation of the functionality of a method and its synchronization conditions, thus resolving inheritance anomalies.
3. For each non-abstract method a single extended (hierarchical) Pr/T-Net [GenLau81] describes the intended functionality.

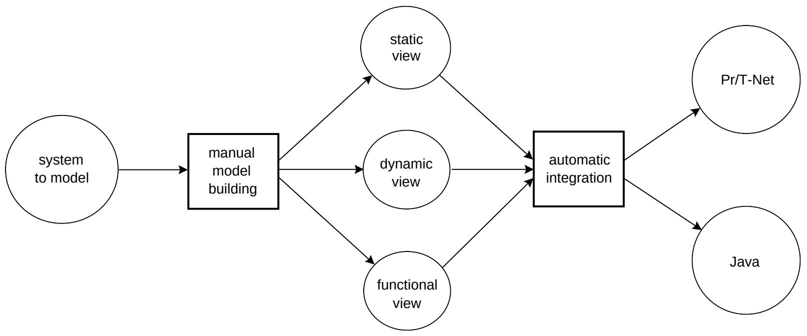


**Fig. 1.** Scenario of OOPr/T-Modeling

Unlike OMT, UML and other basically similar modeling frameworks to set up multiperspective system views, OOPr/T-Models have a formally defined semantics given through a set of rules which allow for the automatic translation of OOPr/T-Models into a single Pr/T-Net as described in section four. Thus, it is not only single views that have a formally defined semantics, but the whole system consisting of different views and their interdependencies does. From a designer's point of view the resulting Pr/T-Net integrating these views should be transparent, because usually only manually created views are visible. The notations to set up these views are described in the following sections.

### 3.1.1 Static view

The first step in modeling states the structure of a system using classes which are related through associations and an inheritance hierarchy. An interface definition can be assigned to each class, consisting of (class) attribute and method signature specifications. To visualize this structure a subset of UML class diagrams is used [Rati99]. Figure 2 shows a class diagram for a producer/consumer system at a low abstraction level.

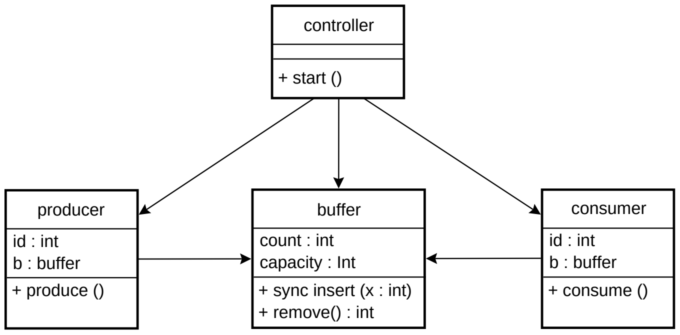


**Fig. 2.** Class diagram for a producer/consumer system

By definition, each system contains a default 'controller' class with a 'start' method as system starting point. The diagram also contains classes 'producer', 'consumer', and 'buffer', the latter associated to the former ones. Class 'buffer' includes two attributes 'count' and 'capacity' for storing actual/maximum data item entries and methods 'insert' and 'remove'. The 'producer' and 'consumer' classes each consist of attributes 'b', implementing the association to 'buffer', and 'id' as well as a single signature for a method without return value. This is an important aspect, as patterns of concurrency are not explicitly defined in OOPr/T-Models in terms of 'threads' or similar low-level concepts. Instead, calls to 'asynchronous methods' not returning any value as well as forward split transitions within functional descriptions of methods start concurrent processes implicitly. Here, asynchronous methods can be synchronized by using an additional keyword ('sync') extending the signature definition if the activation of a method without return value should not lead to the implicit start of a new thread, e.g. method 'insert' of class 'buffer'[3].

### 3.1.2 Dynamic view

The second step in the design process consists of creating a dynamic model for each class of the system. Dynamic models are used to specify activation conditions for publicly available methods. Like attributes, these conditions are object properties, i.e. each object holds its own dynamic model during the runtime of

[3] In contrast, 'synchronized' in Java specifies mutually exclusive access.

a system. A dynamic model is set up using Petri-Nets with anonymous tokens where each transition represents a publicly available method of the corresponding class. To each transition a guard may be assigned containing an expression with attribute identifiers of the associated class. If an object receives a message, the addressed method is activated if the preconditions of the associated transition as well as its guard allow to do so. If a method is activated, tokens within the dynamic model are consumed by the associated transition. After the execution of the method is terminated, new tokens are created on outgoing arcs. Thus, from a designer's point of view transitions within dynamic models have no timeless behaviour, as tokens disappear while methods are being executed. From a semantical point of view, however, this is a syntactical abbreviation, i.e. a shortcut for a more complex 'real' Petri-Net structure with additional places representing currently active methods.

Dynamic models are inherited within a class hierarchy like attributes and methods. Here, dynamic models are only allowed to be modified in subclasses according to refinement rules [Hutten00] based on protocol inheritance [AalBas97].

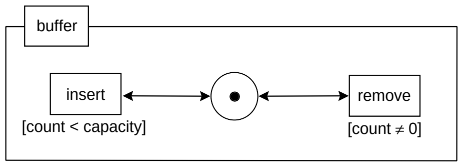


**Fig. 3.** Dynamic model of class 'buffer'

Figure 3 shows the dynamic model for class 'buffer' of the producer/consumer system. Here, methods 'insert' and 'remove' are not allowed to be activated concurrently to avoid inconsistencies. Similarly, 'insert' is only allowed to be activated if the amount of buffer elements is lower than the upper boundary, whereas 'remove' is only allowed to be activated if the buffer is not empty. A closer look on the use of dynamic models to resolve inheritance anomalies is given in section 3.2.

### 3.1.3 Functional view

To be able to model the functionality of a method, high-level Petri-Nets need to be extended to support specific interfacing services. Figure 4 illustrates a method from a black-box point of view, with the following types of interactions with the environment:

1. **An input interface:** As possible input to a method we consider signature specified parameters and current attribute values of the object the respective method belongs to.

2. **An interaction interface:** As the overall behaviour of an object-oriented system is given through the interaction of its message-passing objects, a Petri-Net representing a method has to be able to send messages.
3. **An output interface:** Possible method outputs are new attribute-values of the current object, and return values to the sender of the message which activated the execution of the method.

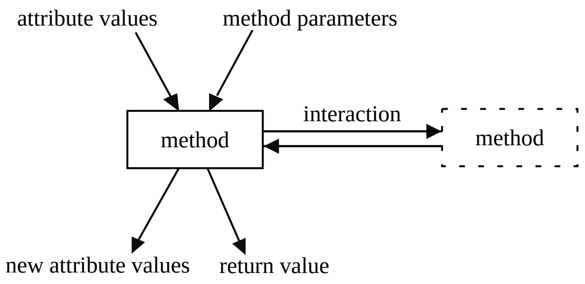


**Fig. 4.** Black-box view of a method

This black-box view of a method does not contain the proper objects as in- and output values. Instead, a finer granularity is given, considering current attribute values. Thus, the answer to the question what exactly flows through a Petri-Net representing a method is not objects, but relevant parts of objects. The reason for the decision not to let the objects themselves flow through a method is to avoid a situation in which one object moves through different concurrent threads of one method, as this could lead to different versions of the same object, each having different attribute values. These versions would have to be merged at the end of the execution of a method with respect to its semantics to become consistent. This would be achieved with the help of additional modeling constructs, which would lead to more complex models.

What follows is a description of Pr/T-Net extensions, introduced to enable communication from a net-specified method with the environment as described above.

**Input interface extensions for Pr/T-Nets:**

The input interface is simply a set of bold printed places. These 'preload places' called net elements serve different tasks. As already mentioned, input values for methods are parameters, given by the message received from the object for which a method is to be executed, as well as current attribute values. Considering a method signature like 'method_name ($\text{arg}_1$ : $\text{type}_1$,...,$\text{arg}_n$ : $\text{type}_n$)', the executing method needs to access parameters $\text{arg}_1$,...,$\text{arg}_n$. Utilizing preload places, import of these parameters into a method is achieved simply by assigning such a place the respective parameter identifier (fig. 5a). If a method is executed, the value of the parameter assigned to the preload place is transparently loaded into this place, which further behaves like an ordinary one. Preload places are used analogously to import attribute values by assigning to them the respective identifiers (fig. 5b), as well as for the initialization of local variables (fig. 5c). Furthermore, 'self' can be assigned to preload places (fig. 5d), which explicitly imports the OID of the object for which the method was activated. Finally, preload places can be annotated with tuples built from these alternatives. In summary, preload places are used to define an object-dependent initial marking for nets describing the functionality of methods in object-oriented models.

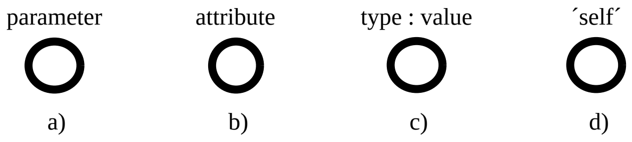


**Fig. 5.** Preload places serving different tasks

**Interacting interface extensions for Pr/T-Nets:**

To communicate with the environment of a method we need to be able to send messages. Therefore, a special kind of transition called 'message transition' is introduced (fig. 6).

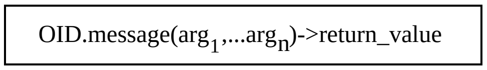


**Fig. 6.** Message transition

From a designer's point of view, a message transition is simply a transition with a message specification in it. The OID of the object the message should be sent to and the respective method parameters have to be transported to the message transition, where arcs can be annotated as usual within Pr/T-Nets. If the method to be activated with a message is a synchronous one, the return value can be used in further steps of the execution of the method. Analogue to dynamic models, such a transition does not fire timeless as it has to wait for the return value of the method to be activated. From a semantical point of view, this is again only a syntactical abbreviation for a more complex net structure.

**Output interface extensions for Pr/T-Nets:**

Similar to the input interface special kinds of places are introduced to serve as output interface. To indicate the end of the execution of a method, a so called 'exit place' is introduced by a double circle (fig. 7a). If a token resides on such a place the execution of the method ends. In case of a synchronous method this token is returned to the sender of the activating message.

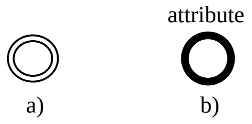


**Fig. 7.** Exit and postsave place

Attributes often have to be updated at the end of the execution of a method. To be able to model such a case in a comfortable way, 'postsave places' are introduced by bold circles like preload places (fig. 7b). Preload and postsave places can be distinguished easily in a given net: the former has at least one outgoing arc and may have incoming arcs, whereas the latter may have incoming arcs only. A postsave place is annotated with the attribute identifier to which the token resident on this place should be the new value as soon as the execution of the method ends. If attributes need to be accessed not only at the start/end of the execution of a method, message transitions calling the implicitly defined 'get' and 'set' methods of an attribute have to be used.

Using Pr/T-Nets with a place capacity of '1' extended this way the specification of methods 'produce' and 'consume' of the example results in fig. 8.

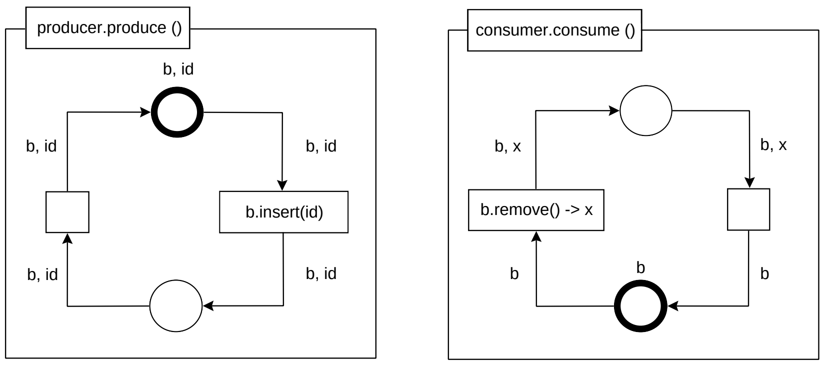


**Fig. 8.** Methods of classes 'producer' and 'consumer'

As 'produce' gives no return value activation of this method results in the (implicit) creation of a new concurrent process. This process imports the current values of attributes 'b' and 'id' into the net, using a single preload place. Afterwards, method 'insert' of the associated 'buffer' object is activated with the 'id' of the producer as argument utilizing a message transition. If the amount of data elements stored within the addressed buffer object exceeds the maximum boundary, or if any of the methods 'insert' or 'remove' is already active, the producer process is suspended. If the state of the dynamic model of the 'buffer' object allows for the activation of 'insert', the suspended producer process continues after termination of this method.

Within method 'consume' a preload place imports the value of attribute 'b' into the net, i.e. the identifier of the associated 'buffer' instance. This value is further used as destination of the message sent to activate the 'remove' method using a message transition. Here, a 'consumer' process implicitly activated by calling its asynchronous 'consume' method is suspended if the state of the dynamic model of the 'buffer' instance demands so, i.e. if there is no element to be removed from the buffer or if a method of this object is currently executed. If the state of the dynamic model of the associated 'buffer' object allows for activation of 'remove', the consumer process continues after termination of this method.

Even if the producer/consumer example is presented including all implementation details, OOPr/T-Models offer support for object-oriented modeling on arbitrary abstraction levels. Class diagrams (which can be organized through packages) to structure a system domain are in principle useful at every abstraction level. Dynamic and functional descriptions can be added and stepwise refined in a seamless way if details become more relevant. The descriptions of the methods of a system may include supertransitions [HuJeSh90] for abstraction purposes in the early stages and functional decompositions in the later ones.

## 3.2 Avoiding inheritance anomalies

After having introduced the OOPr/T-Model notations, the subject of this subsection is now the description of a slightly modified producer/consumer system to illustrate the use of dynamic models to resolve inheritance anomalies [MatYon93].

Inheritance anomalies are very likely to occur within concurrent object-oriented systems if there are no concepts to separate the synchronization conditions of a method from its functionality. In case such concepts are not available, the extension of a superclass by a subclass with additional methods leads to the redefinition of inherited methods within the subclass. The reason for this need to redefine inherited methods stems from the integrated synchronization conditions, which almost inevitably change if a subclass includes additional methods. Due to these problems, early 'object-oriented' programming languages for concurrent systems like POOL/T [Amer87], PROCOL [BosLaf89], and others offered no support for the concept of inheritance to avoid related anomalies.

The separation of a the synchronization conditions of a method from its functionality is realized by OOPr/T-Models through the use of separate dynamic and functional views. To illustrate this aspect, figure 9 gives a modified version of the static view of the initial producer/consumer system.

Here, class 'new_buffer' extends its superclass with an additional method 'remove_new'. This method intends to remove an element out of the buffer only if a call to method 'remove' has not followed the last activation of 'insert', i.e.

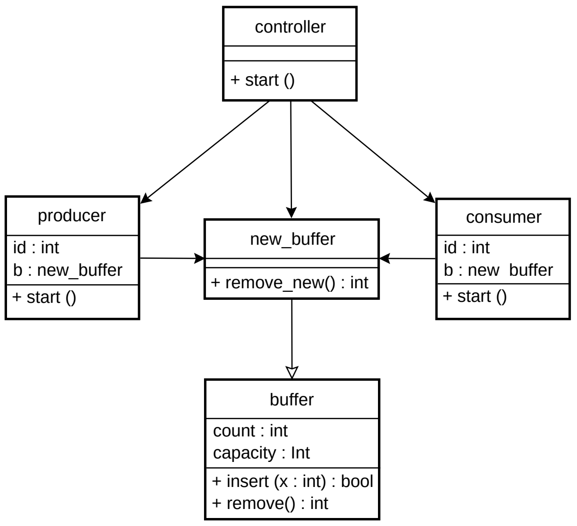


**Fig. 9.** Extended producer/consumer system

'remove_new' is only allowed to be activated immediately after 'insert'. Within a language not integrating concepts to resolve inheritance anomalies, this activation condition leads to a reimplementation of the inherited methods 'insert' and 'remove' to be able to indicate which was activated last, even if there is no need to do so from a functional point of view. OOPr/T-Models avoid such redefinitions as a consequence of class extensions through the use of separate dynamic models, which include synchronization conditions. Figure 10 gives the dynamic model assigned to class 'new_buffer', which specifies that method 'remove_new' is only allowed to be activated if invoked immediately after 'insert'. In consequence, the inherited methods 'insert' and 'remove' remain unchanged in class 'new_buffer', thus resolving inheritance anomalies.

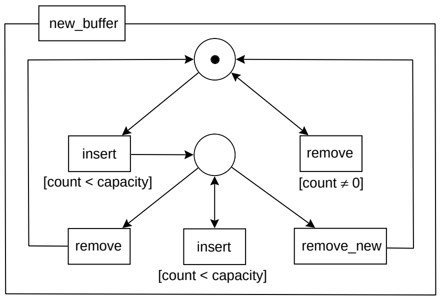


**Fig. 10.** Dynamic model for class 'new_buffer'

## 4 OOPr/T-Models and their formal base

Unlike OMT, UML, and other modeling frameworks to set up multiperspective system views, OOPr/T-Models have a formally defined semantics and allow for the automatic generation of executable Java code. This section outlines both.

### 4.1 Translation to Pr/T-Nets

The formal base of the introduced notation is given by a set of translation rules which generate a Pr/T-Net out of a given OOPr/T-Model. Here, static, dynamic and functional views are integrated. In consequence, it is not only single views but the whole system that has a formally defined semantics in terms of a Pr/T-Net extended with supertransitions and place fusion [HuJeSh90].

The main idea is to use predefined patterns for building a Pr/T-Net-based object-oriented runtime system which integrates the static, dynamic and functional views of an OOPr/T-Modell. The elementary structure of such a Pr/T-Net constructed from a given OOPr/T-Model serves the purpose of message routing, providing the communication infrastructure needed in object-oriented systems. Figure 11 gives an abstract sketch of the runtime system, which is set up hierarchically with a single place as root called 'system message collector'. Every message sent in a system is placed here first, utilizing place fusion. Each class of an object-oriented system is represented by a Pr/T-Net structure, which is connected to the 'system message collector' by a single transition. The guard of such a transition evaluates to 'true' if a message resident on the 'system message collector' is to be sent to the associated class. If such a transition fires, the message is taken from the 'system message collector' to the 'class message collector' of the respective class. The Pr/T-Net representation of each class integrates its associated dynamic and functional models. In order to connect dynamic and functional models to a class representation, the respective views of an OOPr/T-Model need to be transformed into Pr/T-Nets first. This translation, which is in detail described in [Phil99], allows concurrent reentrant usage of functional models, i.e. a Pr/T-Net representing a method exists only once within its defining class. The same holds for dynamic models which are object properties like attributes, but which exist, like methods, only once within each class. If a message is routed to the method to activate, and the dynamic model of the respective object allows for execution, the preload places of the addressed method are initialized.

Figure 12 gives an abstract sketch of the top-level Pr/T-Net structure which represents a class and serves message routing purposes. If a message resides on a 'class message collector', the following cases are distinguished:

- **'create':** In order to instantiate a new object, its definition is needed, i.e. a class representation has to store static information concerning object definitions to be able to instantiate new ones. This is realized using a prototype for each class whose structure reflects the attribute definitions of the static OOPr/T-Model. If a new object is to instantiate, this prototype is duplicated and a new OID generated. This identifier is then returned to the caller

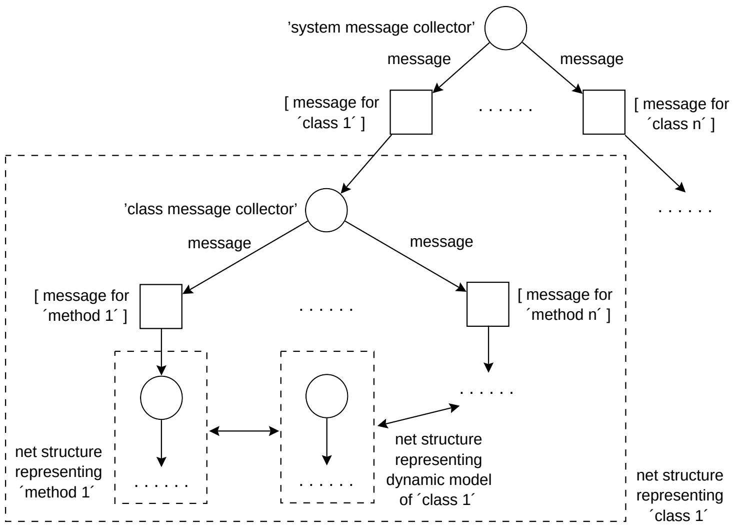

'system message collector'
message
message
[ message for ´class 1´ ]
......
[ message for ´class n´ ]
'class message collector'
message
message
[ message for ´method 1´ ]
......
[ message for ´method n´ ]
......
net structure representing ´method 1´
......
......
net structure representing dynamic model of ´class 1´
net structure representing ´class 1´
......

**Fig. 11.** Abstract sketch of the e.g. system

of the method in order to be able to reference the new object. To store all its objects, each class integrates a single place called 'extent', i.e. if an object is created, the duplicated prototype with its newly generated OID is inserted into the set of already existing objects of this class residing on the extent.

– **'get_attribute', 'set_attribute' :** As the extent storing all objects of a class is transparent, i.e. not directly accessible from a user's point of view, predefined 'get' and 'set' basic update methods for each attribute have to be used. If such a method is invoked, the requested attribute value is extracted/replaced from the respective object in the extent. Due to the fact, that basic update methods are the only way to access attributes of an object, the use of preload and postsave places within functional OOPr/T-Models is only a syntactical abbreviation of their explicit use.

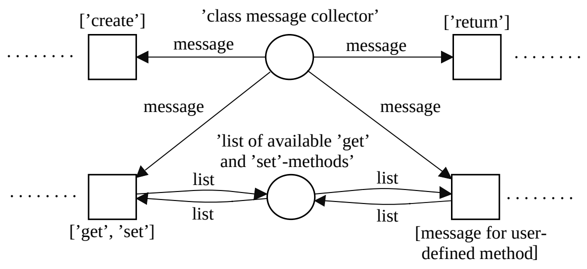


**Fig. 12.** Schematic pattern for representing classes within Pr/T-Nets

– **user-defined methods:** If a message addresses a user-defined method, this message is routed to the Pr/T-Net representation of the respective functional OOPr/T-Model. If the addressed method is inherited, the message to invoke this method is redirected to the superclass where the corresponding Pr/T-Net representation is included. This practice avoids the need to represent the functional model of a method defined in one class within each of its subclasses. If the message reaches the Pr/T-Net representation of the method and the dynamic model allows for execution, the preload places of the method are initialized.
– **return message:** In order to be able to return a value from a synchronous method to the sender of the message activating this method, an internal (transparent) 'return message' is used. If a class receives such a message, the value returned is routed to the respective method.

The Pr/T-Net structures extending the described top-level view of a class representation providing a more detailed insight into the formal base of OOPr/T-Models are given in [Phil99]. A prototype to support the graphical editing of OOPr/T-Models is developed in [George99]. This prototype also allows for the automatic generation of executable Java code, which is described next.

## 4.2 Java code-generation

Due to the formal base of OOPr/T-Models, they are not only suited to serve as architectural layout for implementation. Additionally, executable Java code can be automatically generated, because the formal semantics of OOPr/T-Models as described in the last section is explicitly defined to be compatible with the way object-oriented concepts are integrated in Java (other languages are also possible). This binding to a programming language results from our goal to support seamless software development ranging from high-level analysis to low-level implementation and the not commonly accepted definition of object-orientation, which leads to different programming language interpretations.

At early development stages with their abstract high-level models automatic code-generation is only rarely useful, as implementation details are not known or considered. In contrast, this feature can be useful during the design period to create class frames from the architecture, if a direct use of the destination language for implementation purposes is preferred. If a particular application demands a partial or a complete formal model, OOPr/T-Models can be used down to the visual programming level.

In detail, static, dynamic, and functional views are translated to Java according to the following principles:

– **Static view**: The generation of Java classes from a static OOPr/T-Model is mostly straightforward. A difference between Java and OOPr/T-Models is that the latter supports generic classes (templates). To be able to map this concept to Java, a preprocessing step is introduced which first replaces abstract parameters of generic classes by actual ones. Furthermore, additional classes need to be introduced to the Java code to implement the implicit start of a new Java thread if a method without return value is activated and its 'sync' flag is not set within the OOPr/T-Model.

– **Dynamic view**: A dynamic model specifies activation conditions for publicly available methods of the class it is assigned to. To integrate a dynamic model into each instance of a Java class, each place of such a model is translated into an additional attribute of type 'int', which stores the amount of token resident on the corresponding place. If a publicly available method is to activate, an additional method is called which returns if the current state of the dynamic model allows for the execution of the method. If so, this additional method changes the state of the dynamic model. Then the method is executed and finally the dynamic model is updated again to indicate that the execution of the method is terminated. If the dynamic model does not allow the activation of a method, the requesting thread is suspended until the dynamic model of the particular object changes.
– **Functional view**: The generation of Java methods from corresponding functional OOPr/T-Models is realized with a Pr/T-Net simulator in each method. Here, each place of a functional model is translated into a pair of local variables, the first of which indicates if a token resides on the corresponding place. The particular value of this token is then stored within the second variable. Each transition of a functional OOPr/T-Model is translated into an 'if'-statement as part of a loop which terminates if a value exists on the local variable representing the exit place of the functional OOPr/T-Model. The preconditions to fire a transition are then translated into the enabling conditions of the corresponding 'if'-statement. If such a condition holds, values are removed from local variables representing places with incoming arcs to the respective transition. Furthermore, new values are produced and assigned to the local variables representing places with outgoing arcs from the transition represented by the 'if'-statement.

The described generation of Java code gives reasonable results but is not yet optimized and has several drawbacks which are mainly related to the translation of functional models. In fact, the use of a Pr/T-Net simulator within each method is not the best choice, as only interleaving concurrency is supported within a method. Additionally, this approach lacks efficiency especially considering complex methods. We are currently working on a more sophisticated solution for code-generation which is intended to result in true concurrency within methods and a more efficient code.

## 5 Summary and further perspectives

This article has introduced OOPr/T-Models which were developed to overcome the problems of existing proposals in the area of object-oriented Petri-Nets with respect to a set of properties which we consider essential. As a result of working in this direction, OOPr/T-Models are complete with respect to object-orientation and Petri-Nets. Furthermore, concepts to resolve inheritance anomalies are integrated and a *comparatively* ergonomical notation is given, even if the latter can not be proved due to its qualitative nature. Finally, OOPr/T-Models allow for the multiperspective development of (software) systems with static, dynamic, and functional views on arbitrary abstraction levels ranging from high-level analysis to visual programming. In combination with automatic code-generation, seamless object-oriented software development on a formal base is supported.

OOPr/T-Models have proven to be applicable not only to small systems like the described producer/consumer example which mainly illustrates concurrency synchronization features. An example containing more complex methods from a functional point of view is described in [Phil00] with the specification of a system for the concurrent calculation of primes. More complex concurrent object-oriented systems were created with OOPr/T-Models also, namely a fractal rendering and a ray-tracing system [Hutten00]. The image synthesis of the latter includes features like different geometric objects, multiple lights sources, reflection, shading, transparency etc. All these examples were refined down to the visual programming language level, and executable concurrent Java code was generated from them.

The overall results from developing the described systems using OOPr/T-Models are encouraging, even if there still remain some open issues. To be able to further develop the concepts of OOPr/T-Models and the supporting tool, more case studies from different application areas are needed. Another field of interest is the mapping of UML notations to dynamic and functional OOPr/T-Models. This would allow for the formalization of UML parts and the hiding of Petri-Nets from a designer's point of view, if necessary. Integration of an additional semi-formal (but Petri-Net based) notation to support the communication with domain experts especially in the early analysis stages is possible as well (e.g. [Marx98]).

Besides the ongoing work on these topics, future developments will include extensions of the notation and the tool with concepts to handle persistent data, integration of mechanisms to model/generate distributed systems (CORBA) as well as a GUI building facility. Ideally, these improvements will lead to an integrated CASE-tool for the seamless development of concurrent/distributed object-oriented (software) systems throughout the whole development process on the formal base of Petri-Nets.

## References

[AalBas97] **W.M.P. van der Aalst und T. Basten**. *Life-cycle Inheritance: A Petri-net-based approach. Application and Theory of Petri Nets 1997*, Band 1248 von *LNCS*. Springer-Verlag, Berlin, 1997.

[AleGog91] **A. J. Alencar und J. A. Goguen**. *'OOZE: An Object Oriented Z Environment'*. P. America, *'ECOOP '91: European Conference on Object Oriented Programming'*, LNCS 512. Springer-Verlag, 1991.

[Amer87] **P. America**. *'Inheritance and subtyping in a parallel object-oriented language'. 'ECOOP '87: European Conference on Object Oriented Programming'*, LNCS 276. Springer-Verlag, 1987.

[ArnGos96] **K. Arnold und J. Gosling**. *'The Java Programming Language'.* Addison-Wesley, 1996.

[BaDeMa88] **E. Battiston, F. DeCindio und G. Mauri**. *'OBJSA Nets: a class of high level nets having objects as domains'. 'Advances in Petri-Nets 1988'*, LNCS 340. Springer-Verlag, 1988.

[BoNuFe97] **T. Boehme, J. Nuetzel und W. Fengler**. *'Objektorientiertes Entwurfsmodell für Steuerungssysteme auf Basis der Petri-Netz-Theorie'.* D. Abel E. Schnieder, *'Entwurf komplexer Automatisierungssysteme'*, Braunschweig, 1997.

[BosLaf89] **Jan van den Bos und Chris Laffra**. *PROCOL – A Parallel Object Language with Protocols. Proceedings of the OOPSLA '89 Conference on Object-oriented Programming Systems, Languages and Applications*, S. 95–102, Oktober 1989.

[BruBal86] **G. Bruno und A. Balsamo**. *'Petri net-based object-oriented modelling of distributed systems'.* ACM SIGPLAN Notices, 21(11), November 1986.

[Burk94] **R. Burkhardt**. *'Modellierung dynamischer Aspekte mit dem Objekt-Prozess-Modell'.* Dissertation, Technische Universität Ilmenau, 1994.

[CeJaVo97] **M. Ceska, V. Janousek und T. Vojnar**. *'PN-Talk - A Computerized Tool for Object-Oriented Petri Nets Modelling'. 'Proceedings of the 6th International Workshop on Computer Aided Systems Theory - EUROCAST'97'*, LNCS 1333. Springer-Verlag, 1997.

[Deck95] **G. Decknatel**. *'F-Nets'.* Diplomarbeit, Universität Koblenz-Landau, 1995.

[Durr92] **E. Durr**. *'A formal specification language for object-oriented designs'.* P. Dewilde und J. Vandewalle, *'IEEE CompEuro 92 Proceedings'.* IEEE Press, 1992.

[Engl93] **S. English**. *'Coloured Petri Nets for object-oriented modelling'.* Dissertation, University of Brighton, 1993.

[EnLeRo90] **J. Engelfriet, G. Leih und G. Rozenberg**. *'Net-Based Description of Parallel Object-Based Systems'. 'Foundations of Object-Oriented Languages'*, LNCS 489. Springer-Verlag, 1990.

[Evans*98] **Andy Evans, Jean-Michel Bruel, Robert France, Kevin Lano und Bernhard Rumpe**. *Making UML Precise.* Luis Andrade, Ana Moreira, Akash Deshpande und Stuart Kent, *Proceedings of the OOPSLA'98 Workshop on Formalizing UML. Why? How?*, 1998.

[GenLau81] **H. J. Genrich und K. Lautenbach**. *'System Modelling with High-Level Petri Nets'.* Theoretical Computer Science, 13(1), 1981.

[George99] **T. George**. *'OOPr/T-Modeller : Ein Werkzeug zur Modellierung nebenläufiger objektorientierter Systeme auf der Basis von UML und Petri-Netzen'.* Diplomarbeit, Universität Koblenz-Landau, 1999.

[GiGrWi98] **H. Giese, J. Graf und G. Wirtz**. *'Modeling Distributed Software Systems with Object Coordination Nets'. 'Int. Symposium on Software Engineering for Parallel and Distributed Systems (PDSE'98)'*, Kyoto, April 1998.

[HeeVer91] **K. M. van Hee und P. A. C. Verkoulen**. *'Integration of a Data Model and High-Level Petri-Nets'. 'Proceedings of the 12th Interna-*

*tional Conference on Applications and Theory of Petri-Nets'*, Gjern (Denmark), 1991.

[HolVer95] **T. Holvoet und P. Verbaeten**. *'PN-TOX: a Paradigm and Development Environment for Object Concurrency Specifications'. 'Proceedings of the 16th International Conference on the Application and Theory of Petri-Nets'*, Turin, 1995.

[HuJeSh90] **Peter Huber, Kurt Jensen und Robert M. Shapiro**. *'Hierarchies in Coloured Petri Nets'*. G. Rozenberg, *'Advances in Petri Nets 1990'*, LNCS 483. Springer-Verlag, 1990.

[Hutten00] **P. von Hutten**. *'Modellierung eines Ray-Tracers mit OOPr/T-Modellen'*. Diplomarbeit, Universität Koblenz, erscheint 2000.

[KapSch91] **G. Kappel und M. Schrefl**. *'Using an Object-Oriented Diagram-Technique for the Design of Information Systems'*. H.G. Sol und K.M. van Hee, *'Dynamic Modelling of Information Systems'*. Elsvier Science Publishers B.V. (North-Holland), 1991.

[KeEvRu99] **S. Kent, A. Evans und B. Rumpe**. *UML Semantics FAQ*. A. Moreira und S. Demeyer, *Object-Oriented Technology, ECOOP'99 Workshop Reader*. LNCS 1743, Springer Verlag, 1999.

[Lakos95] **C. Lakos**. *'From Coloured Petri Nets to Object Petri Nets'. 'Proceedings of the 1st Workshop on Object-Oriented Programming and Models of Concurrency'*, Turin, 1995.

[LanHau94] **K. Lano und H. Haughton**. *'A Comparative Description of Object-Oriented Specification Languages'*. K. Lano und H. Haughton, *'Object-Oriented Specification Case Studies'*. Prentice Hall International, 1994.

[Maier97] **C. Maier**. *'Objektorientierte Analyse mit gefärbten Petri-Netzen'*. Diplomarbeit, Universität Hamburg, 1997.

[Marx98] **T. Marx**. *'NetCase : Softwareentwurf und Workflow-Modellierung mit Petri-Netzen'*. Dissertation, Universität Koblenz-Landau, 1998.

[MatYon93] **S. Matsuoka und A. Yonezawa**. *'Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages'*. Research Directions in Concurrent Object-Oriented Programming, 1993.

[Meyer97] **Bertrand Meyer**. *Object-oriented Software Construction*. Prentice Hall, second edition, New York, N.Y., 1997.

[Petri62] **C. A. Petri**. *'Kommunikation mit Automaten'*. Dissertation, Institut für Instrumentelle Mathematik Bonn, 1962.

[Phil99] **S. Philippi**. *'Synthese von Petri-Netzen und objektorientierten Konzepten'*. Dissertation, Universität Koblenz-Landau, 1999.

[Phil00] **S. Philippi**. *'Modeling of concurrent object-oriented systems using high-level Petri-Nets'. Proceedings of the 4th World Multiconference on Systemics, Cybernetics and Informatics (SCI'2000)*, Orlando, USA, 2000.

[Rati99] **Rational Software Corporation**. *'UML-Documentation 1.3'*. 'www.rational.com/uml', 1999.

[Rumb*91] **J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy und W. Lorensen**. *'Object-oriented modeling and design'*. Prentice Hall International, 1991.

[Stulle97] **M. Stulle**. *'Ereignisdiskrete Zustandsrekonstruktion auf der Grundlage objektorientierter Petri-Netz-Modelle am Beispiel flexibler Fertigungssysteme'*. Dissertation, Technische Universität München, 1997.

# An Architecture for Adaptive Planning and Scheduling of Software Processes Using Timed Colored Petri Nets

By

N.C. Narendra and Indradeb P. Pal
Software Engineering Process Group (SEPG)
Hewlett-Packard India Software Operations Ltd.
29 Cunningham Road
Bangalore - 560 052
Email: {ncnaren,ipp}@india.hp.com

**Abstract.** One of the most vexing problems in managing software projects, is the need to appropriately plan and schedule them. Since software projects are process-oriented, this gives rise to the need for planning and scheduling the processes in a software project, so as to be able to meet the project's objectives within the time and cost constraints imposed on the software project. To make matters worse, the parameters of a software project keep changing all the time, requiring the project team to continuously adapt their processes and replan and reschedule their activities constantly.

In this paper, we present a Petri Net based formalism called TCPN or Timed Colored Petri Net, for modeling, planning and scheduling software processes. Our formalism is based on the same formalism in [4], and we show how it can incorporate planning and scheduling algorithms developed outside the Petri Net community; in particular, planning algorithms from [8], and scheduling algorithms from [9]. We also show that it is an overall Planning and Scheduling architecture, and we also show how this can fit into an adaptive process framework such as the one described in [1]. Since the original formalism in [4] was developed for modeling general workflows, we show how it can be adapted to suit the multidimensional nature of software projects.

## 1. Introduction

In any software project, planning and scheduling is one of the most critical problems that the project team could face. The reason for this, is that improper planning and scheduling will cause severe problems later in the software lifecycle, problems that could become impossible to fix. Hence proper planning and scheduling – and appropriate mechanisms for handling risks and unexpected deviations during the course of the project – is crucial for the project.

In this paper, we present a Petri Net based formalism called *TCPN* or *Timed Colored Petri Net*, for modeling, planning and scheduling software processes. Our formalism is based on the Timed Colored Petri Net formalism of [4]. We show how it incorporates planning algorithms from [8], and scheduling algorithms from [9], i.e., planning and scheduling algorithms from outside the Petri Net community. We also show that it is an overall Planning and Scheduling architecture, and we also show how this can fit into an adaptive process framework [1]. Since the original formalism in [4] was developed for modeling workflows in general, we show how it can be adapted to suit the multidimensional nature of software projects.

This paper is organized as follows. We present some preliminary definitions in the next section. In Section 3, our planning algorithm is presented. We show how – once planning is done - scheduling can be done, in Section 4. In Section 5, we demonstrate how our algorithm incorporates the adaptivity that is a crucial aspect of software processes. In Section 6, we present an example from a real-life project that illustrates our ideas. The paper concludes in Section 7 with suggestions for future work.

## 2. Preliminaries

Every software project is composed of the following basic entities:

- *Activities*: These are the steps followed during the software project, and are usually composed into two types of processes in the project:
  - Engineering processes: these are the "regular" processes implemented by the project team in order to develop their deliverables
  - Support processes: these are the processes that are required to be executed by the project team in coordination with central groups such as the SEPG, SQA, etc. Although these may not provide immediate benefit to the software project, they are useful for providing project-level process performance information to the central groups, which can in turn be used to improve future projects
- *Artifacts*: These are the internal and external deliverables produced by the project team.
- *Agents*: These are the roles played by different individuals in the project team, e.g., Project Manager, Technical Lead, Testing Engineer, etc.
- *Resources*: These are the software and hardware resources needed to execute the project, e.g., software tools, special-purpose hardware, etc.

From the above definitions, it is clear that software development is truly a multi-dimensional activity, involving both engineering and support processes, and where the project team needs to balance among activities, artifacts, agents and resources.

Most software projects can be modeled as per a 3-tier architecture, thus:

- The goals of the project and the organization to which it belongs, is the top tier. These goals are usually business-driven, and can be mapped into goals for particular software processes
- The middle tier typically models the usual lifecycles that the software projects follow, e.g., waterfall, V-model, spiral, etc.
- The lowest tier represents the actual processes defined and followed in the project; it is an instantiation of the lifecycle that the project team has chosen, and which has been modeled in the middle tier

In [1], the first author has described a similar 3-tier architecture for general workflow processes, which finds application in software projects also. The corresponding layers from [1] are Planning, Schema and Process Layers, respectively.

We now present the Petri Net definitions. A *Timed Colored Petri Net (TCPN)* (see Fig. 1) is a five-tuple N = (*P,T,I,O,TS*) satisfying the following requirements:

i) *P* is a finite set of places

ii) *T* is a finite set of transitions

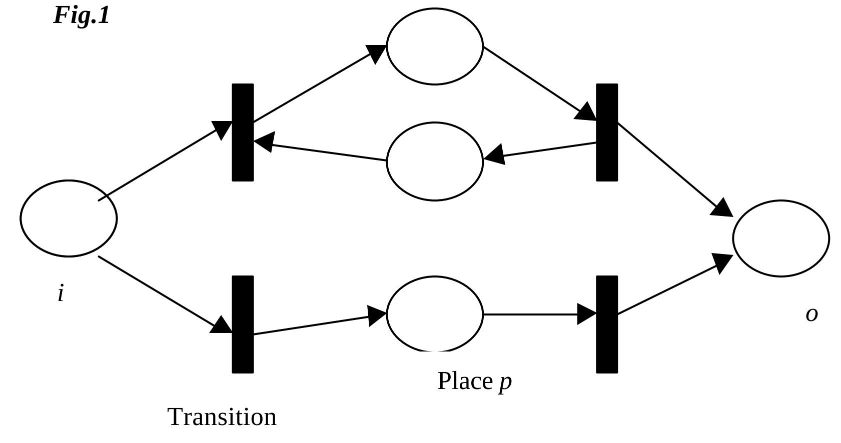
iii) *I* belonging to *T* is the set of input places for each transition, i.e, the pre-set of *T*

iv) *O* belonging to *T* is the set of output places for each transition, i.e., the post-set of *T*

v) *TS* is the time set, i.e., the set of execution times for each transition (expressed as execution intervals)

vi) Also, each place p in P has a set of allowed colors attached to it and this means that a token residing in p must have a value which is an element of this set. In other words, the colors specify the different types associated with each token. In the software project context, each token represents a resource (such as a human, or computer resource) which is consumed during a transition.

***Fig.1***



A marking *M* of the Petri Net represents the state of the Net, i.e., the distribution of tokens over places. Whenever a transition fires, the marking changes, since tokens get redistributed over the Net; hence a transition firing causes a state change in the Petri Net. A marking *M''* is said to be reachable from a marking *M'* if it possible to reach *M''* from *M'* by a sequence of transition firings.

A Petri net is said to be *live* if for every reachable state $M'$ and every transition $t$, there is a state $M''$ reachable from $M'$ which enables $t$.

A Petri net is said to be *bounded* or *k-safe* if and only if for each place $p$ there is a natural number $k$ such that for every reachable state the number of tokens in $p$ is less than or equal to $k$. If $k$ is 1, then the Petri net is said to be *safe*.

Paths connect nodes by a sequence of arcs. Hence a Petri net is *strongly connected* if and only if for every pair of nodes (i.e., places and transitions) $x$ and $y$, there is a path leading from $x$ to $y$.

A *workflow net* (see [6]) is a Petri Net with the following properties:

- It has a unique sink place $o$ (with no output transitions) and a unique source place $i$ (with no input transitions)
- If we add a transition from $o$ to $i$ (hereafter referred to as the *augmented workflow net*), then the resulting Petri net becomes strongly connected.

We call our TCPN representation *well-defined* if and only if it is live and bounded. Hereafter in this paper, we will be dealing only with well-defined TCPNs, due to the following reasons:

- We will see in later sections, that due to our planning and scheduling algorithms, the TCPNs that will be generated, will have to be live
- We will also see in later sections, that the constraints and invariants on the activities and resources of the software project, will ensure that a finite bound can be determined on the resources (and hence, tokens)

A Petri net that is live and bounded, is said to be *sound*. In [6], it has been proved that workflow nets are sound.

***Prop 1***: For our software domain, it is clear that our TCPN must be a workflow net.
***Proof***:

- All software projects should have uniquely defined starting and ending points
- The software processes should be defined so that the resulting augmented workflow net is strongly connected, since there should be a path from any place to any transition in the TCPN representation of the software project's processes

***QED***

***Prop 2***: All well-defined TCPNs are sound.
***Proof***:

From Prop 1, it is clear that the TCPN is a workflow net. From Theorem 1 in [6], a workflow net is sound if and only if its augmented workflow net is live and bounded.

From the above definitions, it is clear that the augmented workflow net for the TCPN is live and bounded, since the underlying TCPN representation is well-defined.
***QED***

A workflow net is said to be *free-choice*, if and only if the following holds:

- For any two transitions, either their presets are identical or they do not have any place in common

Although most workflow models are supposed to be free-choice [Aalst3, pg. 38], TCPNs representing software projects need not be. This is due to the fact that parallel execution threads in a software project may still have dependencies on each other. Our example in Section 6 will illustrate this fact.

## 3. Planning Algorithm

### 3.1 Planning in Software Projects

The task of planning involves the following activities:

- Determining the project requirements and goals – this will involve not only product requirements, but also process requirements (such as, for example, the number of defects that need to be caught during reviews, the maximum number of defects that can be tolerated in any lifecycle phase, or the productivity goals for specified project activities)
- Identifying the resources and staffing available for the project
- Fixing the schedule of the project via negotiations with the customer
- Identifying the major risks in the project [12]

Once these are identified, the planning process basically boils down to determining the project delivery lifecycle, and the different processes that should form part of the lifecycle. This will also involve sequencing the support activities that go along with the lifecycle, such as the following:

- Data collection and submission to the central Software Engineering Process Group (SEPG), who will then do data collation and analysis
- Planning Software Quality Assurance (SQA) activities, such as end-of-phase previews/postmortems, audits, process reviews within the project, in consultation with the central SQA group
- Participation by project team members in organization-wide process improvement activities, in order to support the central SEPG in its activities

These activities should also be executed as per predefined processes, and they also need to be sequenced along with the "regular" project activities.

Since planning is typically a complex and iterative activity involving making choices from among several alternatives, there is a need to encode the relative usefulness of the different choices in terms of the impact that they will make on the overall delivery lifecycle. It is usually convenient to represent these as predicates, in the normal (either conjunctive or disjunctive) form that predicates are usually represented. These predicates are needed either as preconditions or postconditions of activities. Preconditions specify the conditions necessary for an activity to be executed successfully, and postconditions specify the state of the project (from the perspective of that activity) once the activity is successfully executed.

Hence predicates can be specified for the project artifacts that are produced/used during any activity in the software project, and these predicates can be used to derive the appropriate software processes for the project. Some examples are:

- The review should catch at least a minimum number of errors; in other words, the review should have been effective enough to weed out a sufficiently large number of problems in the artifact. Such metrics are typically derived from organization metrics data, and are assigned to the project team by the Senior Management, in the form of quantitative process goals
- There should have been sufficient participation in the review by the project team – in other words, the appropriate team members (decided by skill level, experience, etc.) should have participated in the review

Thus each of these predicates can also be encoded against each of the artifacts in the project, and it will be the responsibility of the project team to ensure that they plan the project activities so that all these predicates are met.

### 3.2 Object-Centered Planning

From the description above, it becomes clear that the planning algorithm that we use, should be "object-centered". In other words, the predicates used in planning should be oriented towards the objects in the projects (viz., agents, artifacts, resources, activities), which will greatly enhance planning efficiency [8].

Hence we have adapted the Object-Centered Planning (OCP) approach presented in [8] for our purposes. The OCP approach consists of the following steps:

- Initial domain description; here, the objects, the sorts (i.e., object classes) that they belong to, the different states and substates that they can exist in for each sort, are described and represented as predicates
- State transition diagrams are then described for each sort in the domain, where each node in the diagram represents a substate class - a disjoint set of substates
- State invariant construction - here, we consider the different ways in which the different sorts can interact with other, and from this we can construct a set of logical invariants for the model. Invariants are nothing but predicates that should always hold during project execution. (more on this in Section 3.3).
- Operator specification; here, parametrized operators that model the effect of actions are specified in terms of the they affect classes of substates

N.B: In the software domain, the operators are nothing but the activities performed by the project team, which can be represented as transitions in the TCPN formalism (this will be described in detail later in this section).

The OCP algorithm basically operates as follows [8]:

- It is a search through a space of partial plans. (The partial plans are nothing but those derived from the project lifecycle model, from which the actual processes can be derived.) First an open node in the set of partial plans is retracted
- If the node does NOT meet the termination condition such that the substates of the node meets a goal condition (i.e., its postconditions as expressed in the form of predicates and invariants do not meet a goal condition), then we find the difference between the objects' current states and their desired states
- An operator (or operator sequence) is then picked that reduces the difference
- If the operator (or operator sequence) is applicable (i.e., its precondition is met), then it is applied to the existing node; otherwise, the weakest precondition of an instantiation of the operator (or operator sequence) is generated, and is used to open a new node
- If the node DOES meet the termination condition, then the node's parent is then opened, the algorithm is then applied on the parent

In the next section, we will describe how this algorithm can be mapped onto our TCPN formalism.

### 3.3 Mapping OCP Onto the TCPN Formalism

As already described above, the OCP formulation easily maps onto the entities in a software project, due to the object-centered nature of OCP. The mapping is given in the table below:

| **OCP** | **Software Project** |
|---|---|
| Sort = object class | Resource class; this includes the classes of the agents (i.e., people performing certain roles in the software project) and classes of the resources (i.e., hardware and software) |
| Objects | Agents and Resources |
| Operator | Change of state as a result of an activity executing – in our TCPN formalism, this represents the execution of a transition |
| Predicate | The predicates can be used to define certain conditions on the resources and how they can be utilized in the project. Each predicate will represent either a precondition or a postcondition on an activity<br><br>These predicates are usually derived |

| | |
|---|---|
| | from the collective experiences of past projects in the organization, and also from historical metrics data which is usually stored in a database [3].<br><br>In our TCPN formalism, these are modeled as places |
| Substate | State of the TCPN at any given time |
| Invariants | Invariants in the software project, i.e., predicates that cannot be violated throughout the project execution – this could be items like resources, cost, etc. |

Using this mapping, it becomes easy to map the OCP algorithm in our TCPN formalism, and obtain a TCPN representation of the software project plan.

The other major aspect of software project planning, and one that has not been considered so far, is risk modeling. Risks are essentially negations of preconditions or postconditions of certain activities in the project plan. Hence, in our TCPN representation, we use the risk modeling method presented in [2] (which is adapted from [12]), and we represent risk modeling as alternate paths in the TCPN representation. In order to do so, we need to consider the following different aspects of risk modeling:

- the *risk factor*, i.e., the characteristic that affects the probability of a risk event occurring
- the *risk event*, which represents the occurrence of a negative incident - or a discovery of information that reveals negative circumstances
- the *risk outcome*, which describes the state of the project after the risk has materialized
- the *risk consequences*, which represents the state of the project after corrective action has been taken
- the *risk effect*, which represents the impact of the risk on the customer and the project
- the *utility loss*, which captures the severity of the loss to the project and to the organization

Hence, we model risks in the following manner:

- determine the risk factors, risk events, risk outcomes and risk consequences of any activity
- model the risk events as negations of preconditions of the activity
- model the risk outcomes as negations of postconditions of the activity
- model the risk consequences, risk effect and utility loss as alternative place-transition sequences in the TCPN representation, in order to deal with the risk (this will be done at the appropriate places which are the pre-sets of the transition representing the activity in question)

## 4.0 Scheduling Algorithm

### 4.1 Introduction to Scheduling

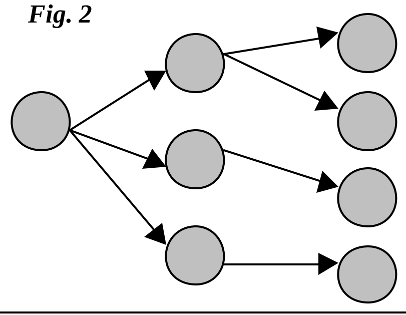
Scheduling basically involves assigning tasks to resources. Typically, before one begins the process of scheduling, the basic assumption is that an initial plan of the project is in place, so that an initial schedule can be drawn up. Hence planning and scheduling inherently follow each other in an iterative fashion, until a satisfactory plan and schedule is reached.

There are typically two basic scheduling approaches [9];

- *Profile-based approaches*: these approaches typically consist of characterizing resource demand as a function of time, and incrementally performing "leveling actions" to (hopefully) ensure that resource usage peaks fall below the total capacity of the resource
- *Clique-based approaches*: given a current schedule, this approach builds a "conflicts graph" whose nodes are activities and whose edges represent overlapping resource capacity requests of the connected activities. Fully connected subgraphs (cliques) are identified and if the number of nodes in the clique is greater than the resource capacity, then we have a conflict

In the context of Petri Nets, some work on Petri Net based scheduling has been done in [4]. This technique uses a Timed Colored Petri Net formalism very similar to ours. In order to represent scheduling, a *reachability graph* is generated. The nodes of this graph are the states in which a Petri Net can exist, and two nodes are connected by a directed edge if one node is reachable from the other by a state change. Hence, reachability graphs can be used to generate feasible schedules. Since there is no time delay for transitions in our TCPN formalism, all our schedules are *eager schedules*, i.e., schedules where resource assignment to tasks happens immediately. Hence, since [4] has shown that the reachability graph can generate all eager schedules, we can use the reachability graph algorithm described therein. In Fig.2, below, we present an example of a reachability graph.

***Fig. 2***



If we map the above concepts to the software domain, we see the following (some of these have already been observed by the authors from their own experiences):

- Conflict detection and resolution are more important than mere resource leveling; this is due to the fact that in software projects, a certain amount of "overloading" is tolerated and sometimes even necessary (due to the demanding, dynamic and semi-chaotic nature of software development)
- Since historical data and past experience data is usually available with most software project teams, either in a database (see [3]) or can be deduced from the past experiences of team members, there is always some level of initial scheduling that can be done by the project team

Hence, we select the clique-based approach, and assume that an initial schedule exists. This schedule is prepared based on the plan derived in Section 3 using the OCP algorithm. Let us assume that this is not consistent, i.e., that there are conflicting requirements on resources.

The generic solver that we use, is based on the one described in [9], and a basic algorithmic template is described briefly here. The template identifies three basic steps that require instantiation:

- *Exists-Unresolvable-Conflict*, which detects an unresolvable conflict,
- *Select-Conflict-Set*, which identifies the set of activities included in the resource conflict to be considered next, and
- *Select-Leveling-Constraint*, which chooses a temporal ordering constraint to solve the conflict by reducing (leveling) resource requirements in conflict.

Before we apply the generic solver to the clique-based approach that we have chosen here, we first describe what a clique is. A clique in a graph $G(V,E)$ is a completely connected subset $C$ of $V$. The size of $C$ is the number of vertices in $C$. The clique in $G$ with maximum size is called the maximum clique. For any vertex $v_i$, the we denote by $J_i$ the set of vertices connected with $v_i$.

Along with the clique information, we need to maintain two graphs for each resource $r_j$. The first graph is the Possible Intersection Graph ($PIG_j$), whose vertices are the activities requiring $r_j$ and whose edges represent the fact that the execution intervals of its two vertices may overlap/intersect in the current solution. The second graph is the Definite Intersection Graph ($DIG_j$), whose vertices are the activities requiring $r_j$ and whose edges represent unresolvable conflicts as described in *Exists-Unresolvable-Conflict*. It is clear that $DIG_j$ is a subset of $PIG_j$.

N.B: The "execution intervals" mentioned are nothing but the starting and ending times for each transition, i.e., the starting and ending times taken for activities in the project, and these are in the time set $TS$ defined in Section 2.

If the resource $r_j$ has capacity $c_j$, then a clique of size at least $c_j + 1$ in the graph $PIG_j$ is called a critical clique in $G$, and represents a potential resource conflict in the current solution.

The algorithm for determining the cliques is as follows:

- Input parameters are the following: a current clique C and a set of vertices $I_D$ used to enlarge the current clique C as the search progresses
- Given $G = (V,E)$ the algorithm starts with $C = \Phi$ and $I_D = V$; this corresponds to the search level $i = 0$.
- At any level $i$ of the search tree, the set $C$ is a clique with $i$ vertices $C = \{v_1, v_2, \ldots, v_i\}$ and the set $I_D$ is obtained by the incremental intersection of $V, J_1, J_2, \ldots, J_i$, where the $J_i$ have been defined above.
- At each step of the algorithm, any of the following conditions hold:
  - ➢ The current clique $C$ has size less or equal to $c_j$, and it is not possible to enlarge it over the threshold - in this case, the search ends in failure
  - ➢ The set $C$ U $\{v_i\}$ is a clique with size greater than $c_j$, in which case the clique with size $c_j + 1$ is collected into the set of cliques
  - ➢ In any other case, the algorithm is recursively invoked on the parameters $C$ U $\{v_i\}$ and $I_{D\text{-}new}$ to check for larger cliques.

The predicate *Exists-Unresolvable-Conflict* is realized by determining the minimal critical sets, i.e, the minimal set of activities that may potentially conflict, using the $DIG_j$. This is done for each resource $r_j$. If no such minimal critical sets exist, then all the conflicts are solvable.

The function *Select-Conflict-Set* is executed by implementing what [9] calls a "least commitment strategy". That is, the set of activities with the least temporal flexibility (i.e., a function of the degree to which the activities can be reciprocally shifted in time) is selected. In other words, the less the temporal flexibility, the more critical it is to resolve first.

The function *Select-Leveling-Constraint* simply chooses the appropriate leveling constraint according to the least commitment strategy described above.

## 4.2 Mapping onto the TCPN Formalism

We now map the scheduling algorithm described above, to our TCPN formalism.

Recall that we have seen that we can use the reachability graph to generate all eager schedules. Hence, each path in the reachability graph from the root to any leaf node represents a possible schedule. The question that we need to answer therefore, is, how to generate the next state from any node in the tree? Since the reachability graph could potentially become infinitely large, the other question to answer is, how to limit the combinatorial explosion?

This is where the scheduling algorithm described in Section 4.1 can be used. The constraints and possible resource conflicts detected during the course of executing that algorithm will limit the number of reachable states from any node in the reachability graph. Hence, the scheduling algorithm of Section 4.1 can be run at every node in the reachability graph, and the next set of reachable states can be derived by not considering those states resulting in conflicts.

## 5.0 Handling Adaptivity in Software Projects

The previous sections of our paper described our TCPN formalism and showed how a general planning and a general scheduling algorithm can be mapped onto our formalism. However, software projects are highly adaptive and semi-chaotic, hence any planning and scheduling architecture for software projects needs to incorporate adaptivity into it.

In [1], we have shown three levels of adaptivity in the workflow context (in increasing order of impact on the software organization), which are also applicable to software projects:

- Adaptivity at process level, i.e., changing certain processes in order to improve project execution
- Adaptivity at lifecycle level, i.e., changing the very delivery lifecycle in order to make substantial changes in the way the software product is developed
- Adaptivity at goal level, i.e., changes in goals resulting in complete re-engineering and re-orientation of the software projects themselves

It stands to reason that adaptivity can be handled efficiently by focussing only on incremental replanning and rescheduling, i.e., only for those portions of the software processes that are actually affected by the change. Hence, our approach to incremental replanning and rescheduling is as follows:

- If the change involves changes in the constraints on the activities, then replanning needs to be looked into first. We first need to determine at what point in the lifecycle the change will begin to affect; we call this the "starting stage". For example, a change in the project requirements may necessitate repeating the design step, or may affect only the coding step. Another example, would be a change in the quality requirements, that may impact only the testing activity, resulting in (perhaps) a minor change to the test plan.

  The next step is to propagate the changed constraints downstream from the starting stage, and determine how much of the rest of the lifecycle is affected by the change. The last activity that is affected by the change, is called the "ending stage".

  In such a situation, we need to rerun the OCP algorithm from the starting stage to the ending stage, with the constraints at the starting stage being the start constraints, and the constraints at the ending stage being the goal conditions of the OCP algorithm.
- If the change does not involve changes in constraints on activities, or if the above step results in changes in constraints on resources, then the scheduling algorithm described in Section 4 needs to be invoked on the portion of the Petri Net that is

between the starting and ending stages. Again, this results in redeveloping the reachability graph between the highest node (in the reachability graph) corresponding to the starting stage, and the lowest node (in the reachability graph) corresponding to the ending stage, as per the algorithm in Section 4.

- The above two steps need to be iterated until a feasible solution is found.

In software projects, the other aspect of adaptivity, is the need to suitably "change-proof" the software lifecycle model, so that future changes to the project are anticipated and taken into account. This can be done using the risk modeling approach described in Section 3.3. As is common in most software organizations, all the most common types of possible risks can be modeled in a risk database (similar to the one described in [3]) and can be used to suitably build in risk management while planning and scheduling. Hopefully, this will minimize replanning and rescheduling.

## 6.0 An Example - Introduction

The example that we have chosen, is simple enough to be used in this paper as an illustration, but it is also derived from a real-life software project in our organization that exhibited all the characteristics that make software project planning and scheduling a complex and demanding activity.

The project, which we will call ABC, is essentially to develop a set of modules that will emulate the behavior of one operating system on another one. The ABC project possesses the following characteristics:

- The requirements are not clearly defined by the customers, who are themselves not very sure of what is to be expected; hence the project team needs to make assumptions which could be invalidated at any time by the customers
- Like any typical software project, ABC’s schedule is quite tight. However, resources are more flexible, since project team members have offered to work overtime if necessary to get the job done, especially since this project is considered to be crucial to the long-term success of the organization.

The project’s lifecycle can be evolutionary, consisting of several basic three-phase waterfall models comprising design, coding and testing. Hence the project consists of design-coding-testing cycles executed one after another. Of course, this is done per module, hence the project lifecycle will be a set of design-coding-testing cycles executed both sequentially and in parallel. Needless to say, since all the modules are supposed to interact with each other, there will also be dependencies/interactions among the 3-phase cycles corresponding to the modules. A representation of the project, for two parallel but interdependent modules, and for one cycle in the overall evolutionary lifecycle, is given in Fig.3 below. Please note the two arrows originating from transitions in Module #2 and ending in places in Module #1 - they depict the dependencies/interactions among the two 3-phase cycles.

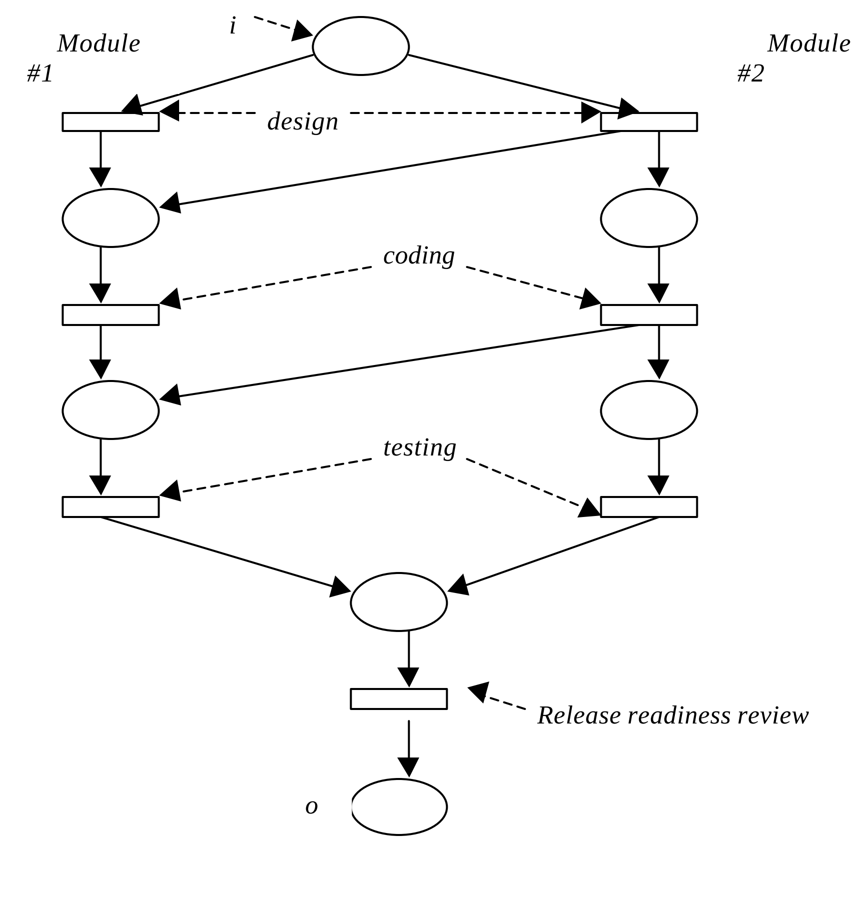

i
Module #1
Module #2
design
coding
testing
Release readiness review
o

**Fig.3**

## 6.1 Planning Algorithm Implementation

The first step in planning the ABC project, is to identify the following, as per the OCP algorithm:

- *Object Classes*: The object classes for our example, are the following
  - Project Manager
  - Engineer
  - Project Quality Interface (he/she is the individual who interfaces with the central SEPG and SQA groups, and helps the PM in coordinating the SEPG and SQA functions in the project team)
- *Objects*: The objects in the ABC project, are the Project Manager (PM) and the 4 engineers
- *Invariants*: There are certain invariants on the activities in the ABC project. Some of them are:
  - All the activities in any 3-phase cycle will have to be executed sequentially
  - Certain dependency invariants exist between modules, which are depicted as constraints. For example, Module #1 cannot be completed until the interface of Module #2 is completed, since Module #2 needs to interact with Module #1 using the interface
  - We can (and in fact, we should) also have invariants derived from historical data from past projects in a historical database [3]; this helps during planning and scheduling, since it minimizes the combinatorial explosion
- *Operators*: These are essentially the activities executed by the project team, which changes the substates of each of the artifacts in the project
- *Predicates*: Predicates are used to denote preconditions and postconditions of an activity. For example, for a coding activity for a module in any 3-phase cycle to start, the following could be the preconditions:
  - The design should have been completed, reviewed and baselined
  - The module's interface dependencies with all other modules are also part of the design which has been reviewed and baselined

  The postconditions could be the following:
  - The code has been reviewed and baselined
  - The code is consistent with the design at the time of coding

- Substates: The substate of the project at any given time, is the substate of its encoded TCPN representation

Basically, the planning algorithm can be implemented as explained in Section 3.2. We start with the design activity in any module (which is an object in our adapted OCP algorithm), and check the extent to which it meets the goal conditions. We note the difference between the goal conditions and the current state of the module, and select an operator (or operator sequence) that reduces the difference. In this case, there could be several operators to choose from (i.e., one or all of the following):

- Coding the module
- Developing the interface to the other modules in the system
- Leaving this module as it is, and designing and/or coding any other module

In case no operator is applicable, we choose the one that generates the weakest precondition (i.e., the operator that violates the predicates the least) and it is used to open a new node in our search space.

So far, we have not mentioned risk management in this planning exercise. Some possible risks to this project are:

- Sudden changes in customer requirements
- Attrition
- Lack of understanding of the domain, which could render the planning estimates useless

These risks are modeled using the modified Riskit methodology described in Section 3.3. This can be done by adding alternate paths in the completed TCPN after the planning algorithm is implemented.

## 6.2 Scheduling Algorithm Implementation

As explained in Section 4.1, we assume that a preliminary schedule exists, and we select the clique-based approach. The search procedure is essentially the one described in [4], and involves generation of the reachability graph. The Possible Intersection Graphs (PIG) and Definite Intersection Graphs (DIG) are used in order to curtail the size of the reachability graph, as described in Section 4.1.

## 6.3 Handling Adaptivity

### 6.3.1 Replanning

Due to the evolutionary nature of the ABC project, it is clear that change is a constant in this project. Since software development is a multi-dimensional activity, the following are some of the changes encountered by the team:

- The customer reorders the priority of certain modules in the system, thus forcing the project team to replan and reschedule from where they were before the reordering
- One of the team members falls ill and is absent for a week, forcing others to make up for this loss. This may not cause a replanning per se, but causes a rescheduling of activities among the existing team members, which in turn results in replanning the activities that the sick team member is supposed to accomplish

- The organizational SQA group announces an unscheduled audit, due to a directive from senior management; this results in significant replanning and rescheduling, since even the 10% overtime is not sufficient for the audit

We will illustrate the first change, i.e., customer reordering, only. In this case, the OCP algorithm must be re-implemented from the stage when the change was ordered. There are essentially three cases here:

- *Completed activities*: since these activities have already been completed, the postconditions of these activities can be used as preconditions for future activities that will be planned
- *Activities yet to be started*: these will be invalidated by the reordered priorities, since they should be replanned afresh
- *Activities in progress*: the fate of these activities has to be decided after the replanning is done. If these activities are to be executed (i.e., if the operator sequence contains these activities), then the project team can simply continue where they left off and complete these activities. Otherwise, the activities will have to be scrapped in favor of the new set of activities.

### 6.3.2 Rescheduling

In the case of rescheduling, the reachability graph needs to be modified from the "starting stage" to the "ending stage", as explained in Section 5. This will also involve modifying the PIGs and DIGs for those resource that are affected. Once again, there are three cases:

- *Resources that are not needed between the "starting" and "ending stages"*; the constraints on the state of the reachability graph at the "starting stage" for these resources remain unchanged
- *Resources that are yet to be used*: these will be invalidated by the change, and will have to rescheduled afresh
- *Resources that are in the process of being us*ed: here, the extent to which these resources have to be reassigned, will depend on the extent of the scheduling change. Some of the activities being executed by these resources may need to be modified - this may in turn trigger a replanning, depending on the effect it will have on the currently running set of activities (i.e., operator sequence as mentioned in Section 6.3.1).

## 7.0 Conclusions and Future Work

In this paper, we presented a Timed Colored Petri Net (TCPN) formalism for representing processes in software projects. We also showed how planning and scheduling algorithms from outside the Petri Net community can be incorporated into our formalism, thereby enhancing the power and usefulness of our formalism. We have also shown how this can also serve as an Adaptive Planning and Scheduling architecture, by aligning it with an adaptive process framework from [1]. We have also illustrated the algorithms with a real-life example from a software project. Although this paper has not described the TCPN formalism itself in any detail, we believe that our major contribution is that we have shown how planning and scheduling algorithms from outside the Petri Net or Software Engineering Community can be used to plan and schedule software processes using Petri Nets.

Our paper brings up several avenues for future work:

- Efficient implementation of our idea, and experimentation with several real-life examples
- A more mathematically rigorous characterization of software processes, including deriving properties similar to those derived in [5] for workflow nets. Also, since Petri Nets are most suited for performance modeling, another open issue is how to appropriately model and analyze the performance of software processes, perhaps using stochastic models [14].
- Modeling of our TCPN formalism in a distributed environment, and implementation thereof; that is, distributing portions of the TCPN and its associated planning and scheduling algorithms among different servers, and the coordination issues resulting therein. Also, how can our ideas be embedded into an agent-based environment such as the one described in [13], where the TCPN for each agent (which will describe the agent's execution methodology) can dynamically adapt to changed circumstances. This also brings up the issue of implementing adaptive planning and scheduling (as in Section 5) among distributed servers. A beginning has been made in [11], but more remains to be done.

## 8.0 Acknowledgments

The authors wish to acknowledge Akshya Prakash, Padma Ravichander and V.S. Subrahmanyam for supporting their work.

## References

[1] N.C. Narendra, “Adaptive Workflow Management – An Integrated Approach and System Architecture,” ACM Symposium on Applied Computing 2000, to appear

[2] N.C. Narendra, “Goal-based and Risk-based Creation of Adaptive Workflow Processes,” American Association for Artificial Intelligence (AAAI) Spring Symposium 2000, to appear

[3] Rajesh Bhave and N.C. Narendra “An Innovative Strategy for Organizational Learning,” World Congress on Total Quality (WCTQ) 2000

[4] W.M.P. van der Aalst, "Petri Net Based Scheduling," Computing Science Reports 95/23, Eindhoven University of Technology, Eindhoven, 1995, also available from http://wwwis.win.tue.nl/~wsinwa/orspec.ps

[5] W.M.P. van der Aalst, "How to Capture Dynamic Change and Management Information? An Approach Based on Generic Workflow Models," Technical Report, UGA-CS-TR-99-01, University of Georgia, Department of Computer Science, Athens, USA, 1999, also available from http://wwwis.win.tue.nl/~wsinwa/genwf.ps

[6] W.M.P. van der Aalst, "Petri-net-based Workflow Management Software", In A. Sheth, editor, Proceedings of the NFS Workshop on Workflow and Process Automation in Information Systems, pages 114--118, Athens, Georgia, May 1996.

[7] Ellis, C., Keddara, K., and Wainer, J., "Modeling Workflow Dynamic Change Using Timed Hybrid Flow Nets", Proceedings of the Petri Net Workshop on Workflow Management: Net Based Concepts, Models, Techniques and Tools, June 1998.

[8] D.E. Kitchin and T.L. McCluskey, "Object-Centered Planning," 15th Workshop of the UK Planning and Scheduling Special Interest Group, Liverpool John Moores University, Liverpool 1996; available from ftp://helios.hud.ac.uk/pub/artform/sig_96.ps

[9] A. Cesta, A. Oddi and S.F. Smith, "Scheduling Multi-Capacitated Resources under Complex Temporal Constraints," Technical Report CMU-RI-TR-98-17, Robotics Institute, Carnegie-Mellon University, 1998; available from http://www.cs.cmu.edu/afs/cs/project/ozone/www/PCP/publications/cl-pro-techrpt.html

[10] C. Cheng and S.F. Smith, "Generating Feasible Schedules under Complex Constraints," in Proceedings of 12th National Conference on AI (AAAI-94), 1994.

[11] T. Bauer and P. Dadam, "Efficient Distributed Control of Enterprise-Wide and Cross-Enterprise Workflows," Proceedings Workshop Enterprise-wide and Cross-enterprise Workflow Management: Concepts, Systems, Applications, 29. Jahrestagung der GI (Informatik '99), S. 25 - 32, 1999; available from http://www.informatik.uni-ulm.de/dbis/papers/1999/BaDa99a.pdf

[12] J. Kontio, D. Getto and D. Landes, "Experiences in improving risk management processes using the concepts of the Riskit method," available from http://mordor.cs.hut.fi/~jkontio/fse6-rm.pdf

[13] Q. Chen, P. Chundi, U. Dayal and M. Hsu, "Dynamic Agents", International Journal of Cooperative Information Systems, 1998

[14] M. Silva and J. Campos, "Structural Performance Analysis of Stochastic Petri Nets," In Procs. IEEE Intern. Computer Performance and Dependability Symp., pp. 61-70, Erlangen, Germany, April 1995; available from http://www.cps.unizar.es/~jcampos/

# Towards Modelling and Verification of Concurrent Ada Programs Using Petri Nets

A. Burns[1], A.J. Wellings[1], F. Burns[2], A.M. Koelmans[2]
, M. Koutny[2], A. Romanovsky[2], and A. Yakovlev[2]

[1] Real-Time Systems Research Group
Department of Computer Science
University of York, U.K.
[2] Asynchronous Systems Laboratory
Department of Computing Science
University of Newcastle upon Tyne, U.K.

**Abstract.** Ada 95 is an expressive concurrent programming language with which it is possible to build complex multi-tasking applications. Much of the complexity of these applications stems from the interactions between the tasks. This paper argues that Petri nets offer a promising, tool-supported, technique for checking the logical correctness of the tasking algorithms. The paper illustrates the effectiveness of this approach by showing the correctness of an Ada implementation of the atomic action protocol using a variety of Petri net tools, including PED, PEP and INA for P/T nets and Design/CPN for Coloured Petri nets.

## 1 Introduction

As high-integrity systems become more sophisticated, the resulting complexity is easier to manage if the applications are represented as concurrent processes rather than sequential ones. Inevitably, the introduction of concurrency brings problems of process interaction and coordination. In trying to solve these problems, language and operating system researchers have introduced new high-level programming constructs. These *design abstractions* are often closely related to the specific domain being addressed. For example, in the world of software fault-tolerance, the notion of conversations [24] and atomic actions [11, 19] are introduced to facilitate the safe and reliable communication between a group of processes in the presence of hardware and software failures, in addition to providing a structuring technique for such systems. Research languages such as Concurrent Pascal have been used as the basis for experimentation [18], or a set of procedural extensions or object extensions have been produced. For example, Arjuna uses the latter approach to provide a transaction-based toolkit for C++ [29]. However, it is now accepted that the procedural and object extensions are unable to cope with all the subtleties involved in synchronisation and co-operation between several communicating concurrent processes.

The main disadvantage of domain-specific abstractions is that they seldom make the transition into general-purpose programming languages or operating systems. For example, no mainstream language or operating systems supports the notion of a conversation [9]. The result is that all the hard-earned research experience is not promulgated into industrial use.

If high-level support is not going to be found in mainstream languages, the required functionality must be programmed with lower-level primitives that are available. For some years now we have been exploring the use of the Ada programming language as a vehicle for implementing reliable concurrent systems [32]. The Ada 95 programming language defines a number of expressive concurrency features [1]. Used together they represent a powerful toolkit for building higher-level protocols/design abstractions that have wide application. For example, [32] recently showed how Ada 95 can be used to implement Atomic Actions. And, as such an abstraction is not directly available in any current programming language, this represents a significant step in moving these notions into general use. An examination of this, and other applications, shows that a number of language features are used in tandem to achieve the required result. Features include:

- Tasks - basic unit of concurrency.
- Asynchronous Transfer of Control (ATC) - an asynchronous means of affecting the behaviour of other tasks.
- Protected Types - abstract data types whose operations are executed in mutual exclusion, and which supports condition synchronisation.
- Requeue - a synchronisation primitive that allows a guarded command to be prematurely terminated with the calling task placed on another guarded operation.
- Exceptions - a means of abandoning the execution of a sequential program segment.
- Controlled types - a feature that allows manipulation of object initialisation, finalisation and assignment.

The expressive power of the Ada 95 concurrency features is therefore clear. What is not as straightforward is how to be confident that the higher-level abstractions produced are indeed correct. As a number of interactions are asynchronous this presents a significant verification problem. The idea of verification using Model Checking with a finite state model (FSM) of an Ada program was first presented in [10]. This method constructed a set of FSMs of individual tasks interacting via channels, and applied analysis of the interleaving semantics of the product of FSMs using the software tool Uppaal. In this paper, we investigate a complementary approach based on Petri nets and their power to model causality between elementary events or actions directly. This can be advantageous for asynchronous nature of interactions between tasks. Petri nets, both ordinary [26] and high level (e.g. coloured nets [17]) offer a wide range of analysis tools to model and verify the logical correctness according to two crucial kinds of properties: (i) safety - an incorrect state cannot be entered (from any legal initial state of the system); and (ii) liveness - a desirable state will be entered (from all legal initial states of the system).

Petri nets have generally been applied to the verification of Ada programs, e.g. [28, 23, 8]. This work has mostly been focused on the syntactic extraction of Petri nets from Ada code in such a way that the verification of properties, such as deadlock detection, could be done more efficiently. To alleviate state space explosion techniques like structural reduction [28] and decomposition [23] of 'Ada nets' have been proposed.

Our research is based on applying Petri nets to model concurrent Ada code, and using Petri nets tools, such as PEP and Design/CPN, to verify its correctness. However, we propose to deal with the unavoidable complexity of the resulting programs within a compositional approach employing a versatile library of design abstractions with well understood and formally verified properties. Confidence in the abstraction can be significantly increased and the development activity itself supported by modelling, simulation and analysis of the dynamic behaviour of the Petri net model; the behaviour can be analysed either by exploring the set of reachable states of the net or its partial order semantics, such as the unfolding prefix. This library can then be used to tackle the verification of complex designs. Thus, while we are ultimately interested in efficient model checking too, the main focus of this paper is on the semantic modelling of salient task interaction mechanisms from Ada 95. To the best of our knowledge, there has been no attempt of using Petri nets to analyse Ada 95 models of Atomic Actions, particularly with ATC and exceptions. However, some work on analysing Ada 95 programs (with ATC, protected objects, and requeue statement) with Petri nets has been recently reported in [15].

This paper is organised as follows. An introduction to model checking based on Petri nets is given in the next section. We use our existing study of Atomic Actions to illustrate the adopted procedure. A simple model is introduced in Section 3 and its refinement in Section 4. Conclusions are presented in Section 5.

## 2 Model Checking using Petri Nets

Model checking is a technique in which the verification of a system is carried out using a finite representation of its state space. Basic properties, such as absence of deadlock or satisfaction of a state invariant (e.g. mutual exclusion), can be verified by checking individual states. More subtle properties, such as guarantee of progress, require checking for specific cycles in a graph representing the states and possible transitions between them. Properties to be checked are typically described by formulae in a branching time or linear time temporal logic [13].

The main drawback of model checking is that it suffers from the combinatorial explosion problem. That is, even a relatively small system specification may, and often does, yield a very large state space which despite being finite requires computational power for its management beyond the effective capability of available computers. To help cope with the state explosion problem a number of techniques have been proposed which can roughly be classified as aiming at implicit compact representation of the full state space of a reactive concurrent system, or at an explicit representation of a reduced, yet sufficient, state space of the system. Examples of the former are algorithms based on the binary decision diagrams (BDDs) [7]. Techniques aimed at reduced representation of state spaces are typically based on the independence of some actions, which is a characteristic feature of reactive concurrent systems, often relying on the partial order view of concurrent computation. Briefly, in a sequential system, it is the actual order of the execution of individual actions which is usually of importance, whereas in a concurrent system the actual order in which, say, two messages were sent and then received may be irrelevant to the correctness of the whole system. Examples include partial order verification [16, 21] and stubborn set method [31]. The partial order view of concurrent computation is also the basis of the algorithms employing McMillan's unfoldings [14], where the entire state space is represented implicitly using an acyclic directed graph representing system's actions and local states.

Model checking is a technique that requires tool support. For Petri nets, there are many tools of different maturity available. These tools are categorised according to many parameters [33]. In our study, we used three relatively mature tools. One is PEP [2, 3], which uses ordinary Place/Transition nets and a number of model checking methods, such as reachability analysis and unfolding prefix. The second one is INA (Integrated Net Analyzer) [27]. The third is Design/CPN [34], which is based on the Coloured Petri nets and has extensive facilities for simulation and occurrence (reachbility) graph analysis.

## 2.1 The PEP Tool

The PEP tool [2, 3, 22] provides a modelling and verification environment based on Petri nets, however, its principal method of inputting large designs is to use a simple concurrent programming language. The tool compiles a program into an internal representation in terms of a 1-safe Petri net which can then be verified for correctness using a variety of techniques, including ones supported by other model checking tools, such as SPIN or SMV. The relevant correctness properties, can be specified in a general-purpose logic notation, such as CTL* or S4. The PEP system incorporates model checkers based on unfolding and structural net theory.

The PEP tool's additional advantage is that it is based on a compositional Petri net model, both P/T-net based and high-level net based [4–6]. It therefore provides a sound ground to develop a compositional model supporting design abstractions.

## 2.2 The INA Tool

The INA tool is an interactive analysis tool which incorporates a large number of powerful methods for analysis of P/T nets. These methods include analysis of: (i) structural properties, such as state-machine decomposability, deadlock-trap analysis, T- and P-invariant analysis, structural boundedness ; (ii) behavioural properties, such as boundedness, safeness, liveness, deadlock-freeness, dynamic conflict-freenes; (iii) specific user-defined properties, such as those defined by predicates and CTL formulas and traces to pre-defined states. These analyses employ various techniques, such as linear-algebraic methods (for invariants), reachability and coverability graph traversals, reduced reachability graph based on stubborn sets and symmetries.

The INA tool uses a combination of interactive techniques, where the user is prompted for various specifications and queries, and file-processing techniques. The basic Petri net file format is compatible with other tools, such as PED and PEP, using Petri net graphical editors.

### 2.3 The Design/CPN Tool

Coloured Petri Nets (CP-nets) are an extension of the basic Petri Net model [17]. A CP-net model consists of a collection of places, transitions, and arcs between these places and transitions. The model contains *tokens* that flow around the model and are stored in the places. The essential feature of CP-nets is that they allow complex data types, i.e. objects, to be attached to the tokens. These objects contain attributes reflecting the system being modelled. The flow of the tokens is determined by so called *guards*, which are conditions, attached to the arcs of the model, that determine whether a transition is allowed to fire. These guards therefore determine the dynamic behaviour of the model; they allow sophisticated behavioural properties to be modelled. The only software tool currently capable of simulating and analysing CP-Net models and generating an executable code (in the ML programming language) is Design/CPN [34]. In the Design/CPN system, guards are specified in ML. Crucially, Design/CPN allows entry of hierarchical models, which greatly aids in the understanding of complex models.

## 3 Model of Simple Atomic Actions

### 3.1 Atomic Actions

An atomic action is a dynamic mechanism for controlling the joint execution of a group of tasks such that their combined operation appears as an *indivisible* actions [19, 25]. Essentially, an action is atomic if the tasks performing it can detect no state change except those performed by themselves, and if they do not reveal their state changes until the action is complete. Atomic actions can be extended to include forward or backward error recovery. In this paper we will focus only on forward error recovery using exception handling [11]. If an exception occurs in one of the tasks active in an atomic action then that exception is raised in *all* processes active in the action. The exception is said to be *asynchronous* as it originates from another process.

### 3.2 Atomic Actions in Ada

To show how atomic actions can be programmed in Ada [32], consider a simple non-nested action between, say, three tasks. The action is encapsulated in a package with three visible procedures, each of which is called by the appropriate task. It is assumed that no tasks are aborted and that there are no deserter tasks [18].

```
package simple_action is
  procedure T1(params : param); -- from Task 1
  procedure T2(params : param); -- from Task 2
  procedure T3(params : param); -- from Task 3
end simple_action;
```

The body of the package automatically provides a well-defined boundary, so all that is required is to provide the indivisibility. A protected object, *Controller*, can be used for this purpose. The package's visible procedures call the appropriate entries and procedures in the protected object.

The body of the package is given below.

```
with Ada.Exceptions; use Ada.Exceptions;
package body action is

  type Vote_T is (Commit, Aborted);
  protected controller is
    entry Wait_Abort(E: out Exception_Id);
    entry Done;
    entry Cleanup (Vote : Vote_t; Result : out Vote_t);
    procedure Signal_Abort(E: Exception_Id);
  private
    entry Wait_Cleanup(Vote : Vote_t; Result : out Vote_t);
    Killed : boolean := False;
    Releasing_cleanup : Boolean := False;
    Releasing_Done : Boolean := False;
    Reason : Exception_Id;
    Final_Result : Vote_t := Commit;
    informed : integer := 0;
  end controller;

  -- any local protected objects for communication between actions

  protected body controller is
    entry Wait_Abort(E: out Exception_id) when killed is
    begin
      E := Reason;
      informed := informed + 1;
      if informed = 3 then
        Killed := False;
      informed := 0;
    end if;
  end Wait_Abort;

  entry Done when Done'Count = 3 or Releasing_Done is
  begin
    if Done'Count > 0 then
      Releasing_Done := True;
    else
      Releasing_Done := False;
    end if;
  end done;

  entry Cleanup (Vote: Vote_t;
    Result: out Vote_t) when True is
  begin
    if Vote = Aborted then
      Final_result := Aborted;
    end if;
    requeue Wait_Cleanup with abort;
  end Cleanup;

  procedure Signal_Abort(E: Exception_id) is
  begin
    killed := True;
    reason := E;
  end Signal_Abort;

  entry Wait_Cleanup (Vote : Vote_t; Result: out Vote_t)
        when Wait_Cleanup'Count = 3 or Releasing_Cleanup is
  begin
    Result := Final_Result;
    if Wait_Cleanup'Count > 0 then
      Releasing_Cleanup := True;
    else
      Releasing_Cleanup := False;
      Final_Result := Commit;
    end if;
  end Wait_Cleanup;
end controller;

procedure T1(params: param) is
  X : Exception_ID;
  Decision : Vote_t;
begin
  select
    Controller.Wait_Abort(X);
    raise_exception(X);
  then abort
    begin
```

```
        -- code to implement atomic action
        Controller.Done; --signal completion
      exception
        when E: others =>
          Controller.Signal_Abort (Exception_Identity(E));
      end;
    end select;
  exception
     -- if any exception is raised during
     -- the action all tasks must participate in the recovery
     when E: others =>
       -- Exception_Identity(E) has been raised in all tasks

       -- handle exception
       if handled_ok then
         Controller.Cleanup(Commit, Decision);
       else
         Controller.Cleanup(Aborted, Decision);
       end if;
       if Decision = Aborted then
         raise atomic_action_failure;
       end if;
  end T1;


  procedure T2(params : param) is ...;
  procedure T3(params : param) is ...;

end action;
```

Each component of the action (*T1*, *T2*, and *T3*) has identical structure. The component executes a select statement with an abortable part. The triggering event is signalled by the *controller* protected object if any component indicates that an exception has been raised and not handled locally in one of the components. The abortable part contains the actual code of the component. If this code executes without incident, the *controller* is informed that this component is ready to commit the action.

If any exceptions are raised during the abortable part, the *controller* is informed and the identity of the exception passed.

If the *controller* has received notification of an unhandled exception, it releases all tasks waiting on the *Wait_Abort* triggering event (any task late in arriving will receive the event immediately it tries to enter into its select statement). The tasks have their abortable parts aborted (if started), and the exception is raised in each task by the statement after the entry call to the controller. If the exception is successfully handled by the component, the task indicates that it is prepared to commit the action. If not, then it indicates that the action must be aborted. If any task indicates that the action is to be aborted, then all tasks will raise the exception *Atomic_Action_Failure*. Figure 1 shows the approach using a simply state transition diagram.

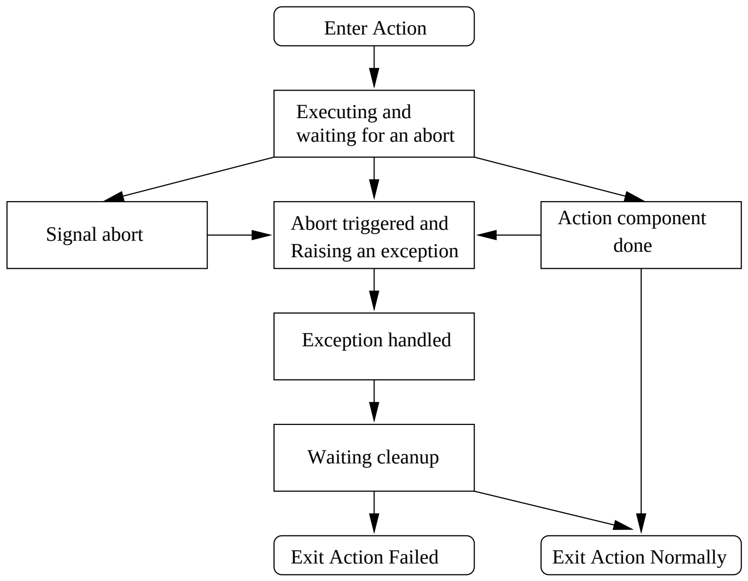


**Fig. 1.** Simple state transition diagram illustrating Atomic Action with forward error recovery for the system with two tasks

### 3.3 Modelling the Ada Implementation in P/T nets

We now consider Petri nets for this Ada code. For the sake of simplicity, we consider only two tasks here. We first look at ordinary P/T nets, i.e. nets without token typing. Each of the client tasks will have an identical PN, specialised only in its labelling of transitions and places. The controller will also be modelled as a single Petri net. Graphical capturing of Petri nets is done using Petri net editor PED [20], which allows hierarchical and fragmented construction of P/T nets, and export to an extensive range of formats including those accepted by analysis tools such as PEP and INA. Figure 2 presents the task model (a) and the controller model (b).

Places and transitions which are not shaded, such as start1 and arr1 are individual for the task net (we show the net for Task 1). Those places and transitions which are shaded are so called logical places and transitions – they are used to interconnect subnets to form larger nets. In other words, by declaring places or transitions in different subnets as logical, we virtually merge such places and transitions in the overall net provided that they have the same label, e.g. waitAbort and sigAbort1. Note that the net models use test or read-only arcs,

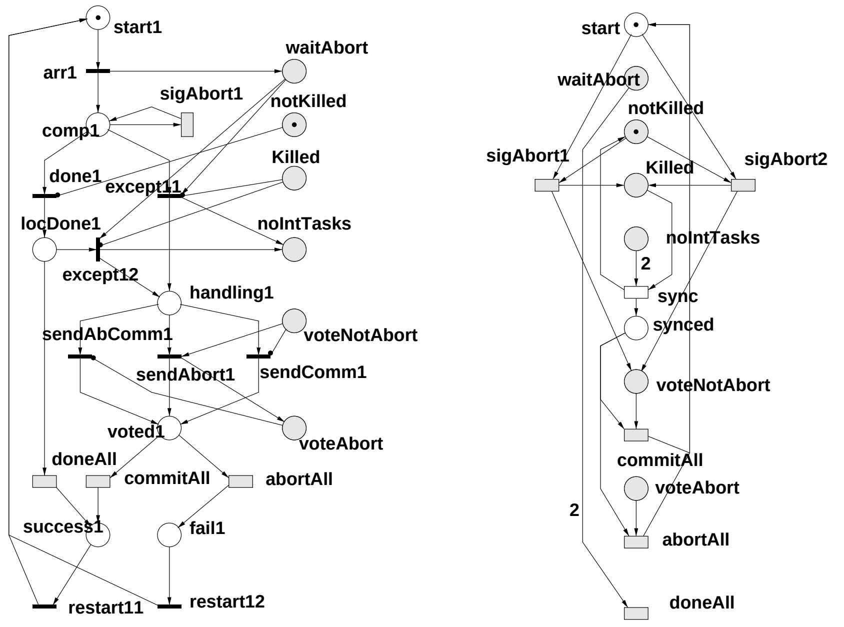


**Fig. 2.** P/T net models: (a) Task model (b) Controller model

which are represented graphically by arcs with a black dot at the transition end, and weighted arcs. The former are used to show the fact that transitions in the task net can test the state of shared variable, such as *Killed*, which is modelled by two complementary places notKilled and Killed in the controller net.

Our basic idea of modelling the Ada code for the Atomic Action behaviour with P/T nets is as follows. We represent states of each task as unshaded places and key actions local to the task as unshaded transitions. Arriving in the Atomic Action by the task is represented by transition arr1. This also generates a token in the place waitAbort, which belongs to the controller and counts the number of tasks that have actually entered the Atomic Action. The place labelled comp1 corresponds to the state of the task in which the task performs normal computation. From this state the task may either: (a) execute transition done1 and go to the Local Done state of normal completion of the action (place locDone1), or (b) it may raise an exception by firing transition sigAbort1 (this corresponds to executing the *Signal_Abort* procedure, which switches the state of the *Killed* flag from *false* to *true* – a token is toggled from place notKilled to Killed), or (c) it may be forced to go to the Error-Handling state (place handling1), either from the Normal Computation state or from the the Local Done state because of some tasks (including itself) has raised an exception, in which case transition except12 will be fired.

Subsequent action of the task depends on whether the task ends in the Local Done or in the Error-Handling state. If the former, the task provides a condition for the controller to fire a shared transition doneAll (corresponding to the execution of the *Done* entry by all tasks). If the task is in the Error-Handling state, it handles the exception, and, depending on the result of the handling, votes either for Action Commit or Action Abort.

The voting mechanism used in Atomic Actions allows one task voting for Abort to force the entire operation into Failure. In our Petri net model, this is achieved by using three transitions sendAbort1, sendComm1 or sendAbComm1, individual to the task. These transitions are connected to two complementary places voteNotAbort and voteAbort in the controller net. Initially, when the voting begins, a token is assumed to be placed into place voteNotAbort. While none of the tasks vote for Abort, the token remains in this place, and if the task votes for Commit, which corresponds to the *handling_ok* flag being set in the task, transition sendComm1 fires due to the reading arc from place voteNotAbort. As soon as one of the tasks votes for Abort, transition sendAbort1 is fired, which toggles the token from voteNotAbort to voteAbort in the controller. This corresponds to assigning the state of the global flag *Final_result* to *aborted* in the *Cleanup* entry. After that, in all tasks, regardless of their individual voting, transition sendAbComm1 will fire due to the reading arc from place voteAbort.

Voting is complete when the task is in the state where it is ready to check the value of the *decision* flag. This corresponds to a token in the voted1 place. At this point all tasks synchronise on firing shared transitions commitAll or abortAll, which are respectively preconditioned by the controller's places voteNotAbort and voteAbort. If the former fires it puts a token in the local success1 place, otherwise the local fail1 is marked. The task subsequently fires one of the two possible restart transitions which corresponds to bringing the task to the state where it is ready to execute the Atomic Action again.

Using the PED tool we constructed the model of the system from the task and controller fragments. Once the appropriate places and transitions are merged the actual behavioural interaction between task and controller is achieved through the following two main mechanisms:

- (i) synchronisation on shared transitions, which is similar to rendez-vous (blocking) synchronisation, and
- (ii) communication via shared places, which is similar to asynchronous (non-blocking) communication.

### 3.4 Verification of the P/T-net model

This P/T net model of the Ada code can be exported from PED to analysis tools, such as INA or PEP. We used PEP, in which we could simulate the token game and perform reachabilty analysis to verify by Model Checking the key properties of the algorithm. First, if 'Task1' is in place success1 then it must not be possible for any of the other tasks (say 2) to be in fail2. This is presented to the reachability analysis tool by the following logic statement:

```
success1,fail2
```

This test gives the `<NO>` result, i.e. such a marking in which these two places are marked is not reachable.

Similarly, to the test for reachability of a marking in which both tasks end in success state:

```
success1,success2
```

the tool reacts with `<YES>` and produces:

```
_SEQUENCE:
arr2,done2,arr1,done1,doneAll
```

which is a firing sequence leading to the global success state.

When setting the option `Calculate all paths` to true, the tool produces the following list of firing sequences:

```
_SEQUENCE:
arr2,done2,arr1,done1,doneAll
arr2,arr1,done2,done1,doneAll
arr1,arr2,done2,done1,doneAll
arr1,done1,arr2,done2,doneAll
arr2,arr1,done1,done2,doneAll
arr1,arr2,done1,done2,doneAll
```

This set, however, includes only those paths which go through the locDone states, but not those which are the result of succcesful handling and overall Commit voting. This is caused by the fact the system searches for all paths satisfying the shortest length criterion.

The effect of a coherent error handling can be tested by:

```
fail1,fail2
```

This results in:

```
_SEQUENCE:
arr1,done1,arr2,sigAbort2,except21,sendAbort2,except12,sendAbComm1,sync,abortAll
arr2,arr1,done1,sigAbort2,except21,sendAbort2,except12,sendAbComm1,sync,abortAll
...
```

all together over 600 paths. These assertions imply inconsistency is not possible.

We have also used tool INA to verify the various behavioural (safety and liveness) properties. The results of this analysis are:

```
Safety Properties:
Safe - No
Bounded - Yes
Dead State Reachable - No
Covered by Transition-Invariants - Yes
```

These results mean that several tasks can enter the controller simultaneously, but that the total number of tasks is bounded. All transitions belong to a transition-invariant, which means that the net is structurally live, i.e. it is sufficiently rich in connections to make it live.

```
Resettable, reversable (to home state) - Yes
Dead transitions exist - No
Live - Yes
Live and Safe - No
```

The computed reachability graph has 76 states.

The INA tool allows to state properties in the form of CTL (Computational Tree Logic) [12] formulas. We can formulate properties of interest, such as whether there exists a path which leads to a state where one task ends in success while the other in fail:

```
 EF((P18&P21 )V(P19 &P20 ))
```

Here P18 (P19) stands for success1 (success2) and P21(P20) for fail2 (fail1). The result of the check is:

```
 s1 sat EF((P18 &P21 )V(P19 &P20 )):FALSE
```

Another interesting property would be, whether there is a path that leads to a state in which both tasks end in success but the flag Killed (place P7 below) has been set to true:

```
 s1 sat EF(P7&(P18 &P19 )): FALSE
```

For comparison, we have tried a modified net model for a task – we omitted a read arc leading to transition done1 which tests flag notKilled. This modification may correspond to allowing the code for a task to be non-sequential – a task may signal abort and at the same time pass to Local Done (the effect of inertia or delay in reacting to the abort). Interestingly, such a modification does not lead to the violation of deadlock-freeness or the property of both tasks ending either in success or fail. But for the last property above it returns:

```
 s1 sat EF(P7 &(P18 &P19 )): TRUE
```

## 4 CPN Modelling and Analysis

We modelled Atomic Actions using Coloured Petri nets (CPNs) and analysed the model using the Design/CPN tool. The three main CPNs for the model are shown in Figures 3, 4 and 5.

They capture the system hierarchically, as a composition of the controller and task nets. Due to the ability of CPNs to distinguish objects by their token colours and values, we can use the same net structure for all tasks and encode

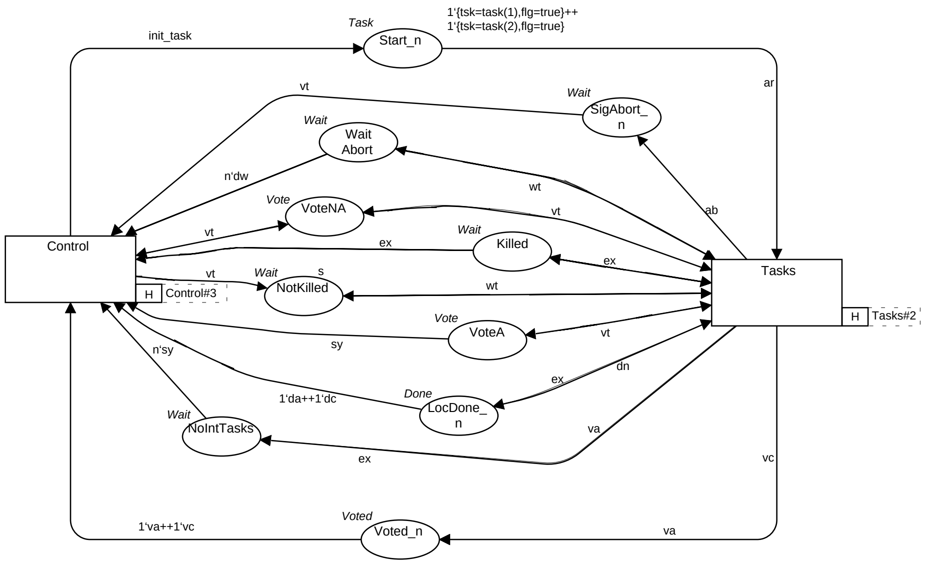


**Fig. 3.** CPN model of the Atomic Action: Top Hierarchy level

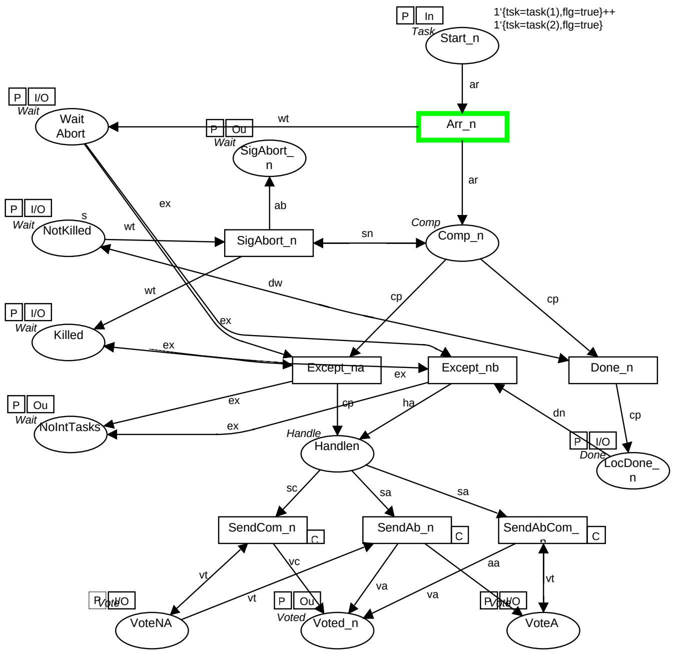


**Fig. 4.** CPN model of Tasks

individual tasks simply by their token values. Another advantage of this type of modelling is that we can parameterise a system model with $n$ tasks and analyse it for different number of tasks by simply setting the $n$ parameter to an appropriate value.

The list of colour definitions (with parameter $n = |Tasks|$ set to 2) is:

```
val n=2;
color Flag = bool;
color Taskn = index task with 1..n;
color Task = record tsk : Taskn * flg : Flag;
color Signal = with s;
var ar, re, ts, cp, sn, sn_, va, vc : Task;
var ca, vt : Signal;
var ab, dw, ex, sy, wt, aa, te : Signal;
var ha, sa, sc, da, dc, dn : Task;

val init_task = 1`{tsk=task(1),flg=true}++
1`{tsk=task(2),flg=true};
```

Here we show the results of the analysis using Design/CPN. Most of the statistics we produced using functions directly available from the Design/CPN menus.

From the *Statistics* it can be seen that the O-graph (Occurrence Graph, i.e. reachability graph) for $n = |Tasks| = 2$ has 63 nodes and 114 arcs. The number of strongly connected components (Scc-graph) are less than the O-graph nodes, implying that an infinite occurrence sequence exists.

```
Statistics
----------
  Occurrence Graph            Scc Graph
    Nodes:  63                    Nodes:  13
    Arcs:   114                   Arcs:   14
    Secs:   0                     Secs:   0
    Status: Full
```

For the *Boundedness Properties* Integer bounds are as expected. The Signal nodes can never be more than one and the Task nodes never exceed two. We also show some of the best Upper Multi-set Bounds to show the task and signal distribution. The best Lower Multi-set Bounds are all empty.

```
Boundedness Properties
-------------------------------------------
  Best Integers Bounds    Upper      Lower
  Control'Fail_n 1       1          0
  Control'Killed 1       1          0
  Control'LocDone_n 1    2          0
 ...

Best Upper Multi-set Bounds:
  Control'Fail_n 1    1`s
  Control'Killed 1    1`s
  Control'LocDone_n 1 1`{tsk=task(1),flg=true}++1`{tsk=task(2),flg=true}
  Control'NoIntTasks 1  2`s
...
```

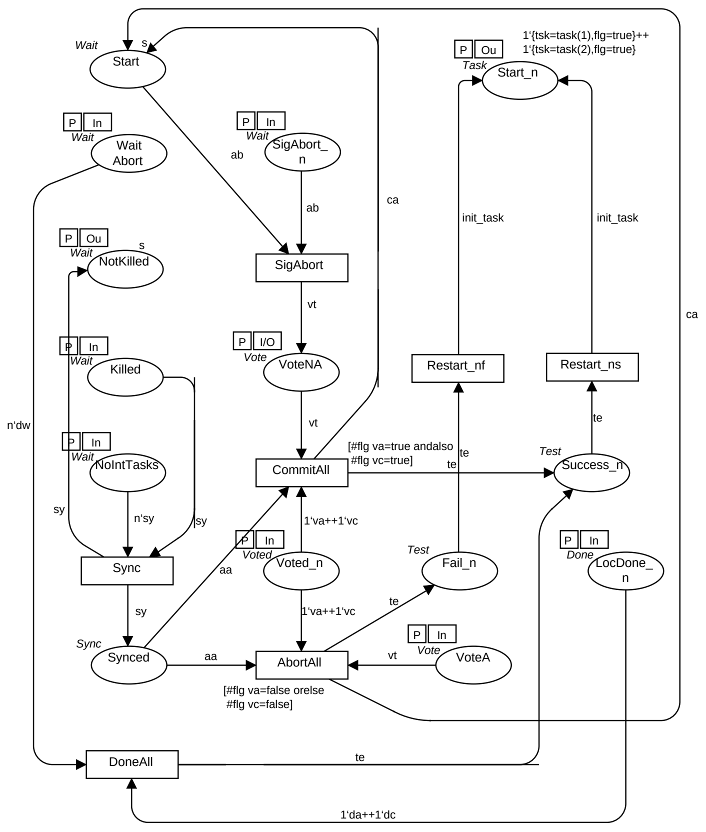


**Fig. 5.** CPN model of Controller

The *Home Properties* show that it is possible to reach any marking from any other marking in the O-graph. The *Liveness Properties* show there are no Dead Markings. Some of the *Fairness* Properties of the O-graph are shown below. Only `Arr_n` is Impartial which implies that repeated cycles of the whole graph require occurence of of firing of this node.

```
Liveness Properties                  Fairness Properties
-------------------------------      -------------------------
  Dead Markings:  None                 Control'AbortAll 1     Fair
  Live Transitions Instances:          Control'CommitAll 1    Fair
```

```
                                     Control'DoneAll 1      Fair
  Control'AbortAll 1                 Control'Restart_nf 1   Fair
  Control'CommitAll 1                Control'Restart_ns 1   Fair
  Control'Restart_nf 1               Control'SigAbort 1     Fair
  Control'Restart_ns 1               Control'Sync 1         Fair
  Control'SigAbort 1                 Tasks'Arr_n 1          Impartial
  Control'Sync 1                     Tasks'Done_n 1         Just
  ...
```

The following are examples of the testing of more specific properties formulated as *Queries* to O-graph and its nodes. These queries are based on functions that are defined in ML.

Function `Success_` tests all markings in which the `Success_n` node is active. A function `Fail_` can be defined in a similar manner.

```
Function
---------------------------------------
fun Success_ (s: Test) : Node list
= PredAllNodes (fn n =>
cf(s, Mark.Control'Success_n 1 n) > 0);
-----------------------------------------
```

The following tests can be run using these functions:

```
Test                                         Result
----------------------------------------     ------------------------------
Success_(s);                                 val it = [29] : Node list
Fail_(s);                                    val it = [63] : Node list
Success_(s) <> Fail_(s);                     val it = true : bool
length(Success_(s))+length(Fail_(s))=2;      val it = true : bool
-----------------------------------------    ------------------------------
```

This means that `Success` and `Fail` do occur, that they cannot occur simultaneously, and that there can be only one of each. All these results are as expected.

We can test for specific occurrences of the `Success_n` node (node 29) to be activated. It shows that there are only two possible occurrences that can lead to this happening, i.e. one from `Voted` causing `Commitall` (node 60) or `Doneall` (node 20).

```
Functions and Tests                          Result
-----------------------------------------    ------------------------------
Success_(s);                                 val it = [29] : Node list
InNodes(29);                                 val it = [60,20] : Node list
OutNodes(60);OutNodes(20);                   val it = [29] : Node list
                                             val it = [29] : Node list
-----------------------------------------    ------------------------------
```

State or occurrence 60 represents transition `CommitAll` being activated which leads to `Success_n`. State or occurrence 20 represents transition `DoneAll` being activated which also leads to `Success_n`. There are no other such occurrences.

Finally, the following table shows how the Occurence graph increases as the number of Tasks is increased.

Size of Occurence Graph with number of Tasks

| \|Tasks\| | Nodes | Arcs | \|Tasks\| | Nodes | Arcs |
|---|---|---|---|---|---|
| 2 | 63 | 114 | 5 | 7568 | 25883 |
| 3 | 298 | 689 | 6 | 39331 | 158444 |
| 4 | 1481 | 4220 | 7 | 207667 | 969677 |

## 5 Conclusion

We have shown that a relatively complicated Ada program using tasking can be modelled and verified using Petri nets (ordinary P/T nets and Coloured) and Model Checking. This significantly improves confidence in the correctness of higher-level abstraction such as atomic actions.

This paper is a preliminary attempt in pursuing our chosen direction of research, in which we would like to develop a more comprehensive methodology for verifying high-integrity systems built of Atomic Actions and implemented in Ada 95.

The major new aspects of this work, which also reveal the potentially exploitable advantages of the Petri net approach over the State Machine one [10], are:

– Refinement of both states and transitions;
– Analysis of behaviour at the true concurrency and causality level;
– High-level aspects of modelling, such as parametrisation, are possible using high-level Petri nets.

For example, if refinement with threads (e.g., task spawning), recursive atomic actions, etc. were possible in the modelled systems, then Petri nets would provide a much more efficient way of modelling than state machines.

We have only shown ways of modelling interaction mechanisms at the semantical level. Part of the intended future work would be to develop new methods of extracting Petri nets from the Ada 95 syntax.

Although this paper has not introduced real-time issues, the choice of tool and modelling technique implies that the approach can be extended to a timed Petri net approach.

## References

1. Ada 95: Information technology - Programming languages - Ada. Language and Standard Libraries. ISO/IEC 8652:1995(E), Intermetrics, Inc., 1995.

2. E.Best: Partial Order Verification with PEP. Proc. of POMIV'96, Partial Order Methods in Verification. G. Holzmann, D. Peled, V. Pratt (eds), Am. Math. Soc. (1997) 305-328.
3. E.Best and B.Grahlmann: PEP - more than a Petri Net Tool. Proc. of Tools and Algorithms for the Construction and Analysis of Systems, 2nd International Workshop, TACAS'96, Passau, March 1996, T. Margaria, B. Steffen (eds). Springer-Verlag, Lecture Notes in Computer Science 1055, Springer-Verlag (1996) 397-401.
4. E.Best, R.Devillers, J.Hall: The Petri Box Calculus: a New Causal Algebra with Multilabel Communication. Advances in Petri Nets 1992, Lecture Notes in Computer Science 609, Springer-Verlag (1992) 21-69.
5. E.Best, R.Devillers, and M.Koutny: Petri Nets, Process Algebras and Concurrent Programming Languages. Lectures on Petri Nets II: Applications, Advances in Petri Nets. Lecture Notes in Computer Science 1492, Springer-Verlag (1998) 1-84.
6. E.Best, H.Fleischhack, W.Fraczak, R.P.Hopkins, H.Klaudel and E.Pelz: M-nets: An Algebra of High-level Petri Nets, with an Application to the Semantics of Concurrent Programming Languages. Acta Informatica 35 (1998) 813-857.
7. R.E.Bryant: Symbolic Boolean Manipulation with Ordered Binary-decision Diagrams. ACM Computing Surveys 24 (1992) 293-318.
8. D.Buchs, C.Buffard and P.Racloz: Modelling and Validation of Tasks with Algebraic Structured Nets. Proc. of Ada in Europe'95, Lecture Notes in Computer Science 1031, Springer-Verlag (1995) 284-297.
9. A.Burns and A.J.Wellings: Real-Time Systems and Programming Languages (Second edition) Addison Wesley (1996).
10. A.Burns and A.J.Wellings: How to Verify Concurrent Ada Programs - The Application of Model Checking. Ada Letters, Volume XIX, Number 2 (1999) 78-83.
11. R.H.Campbell and B.Randell: Error Recovery in Asynchronous Systems. IEEE Transactions on Software Engineering SE-12 (1986) 811-826.
12. E.M.Clarke and E.A. Emerson: Synthesis of synchronization skeletons for branching time temporal logic. In Dexter Kozen, editor, *Logic of Programs: Workshop*, LNCS, vol. 131, Springer-Verlag, 1981.
13. E.M.Clarke and J.Wing: Formal Methods: State of the Art and Future Directions. Report, Carnegie Mellon University (June 1996).
14. J.Esparza: Model Checking Based on Branching Processes. Science of Comp. Prog. 23, 151-195 (1994).
15. R.K. Gedela and S.M. Shatz. Modeling of advanced tasking in Ada-95: a Petri net perspective. Proc. 2-nd Int. Workshop on Software Engineering for Parallel and Distributed Systems (PDSE'97), Boston, MA, pp. 4-14 (May 1997).
16. P.Godefroid and P.Wolper: A Partial Approach to Model Checking. Information and Computation, 110(2), 305-326 (1994).
17. K.Jensen:Coloured Petri Nets. Basic Concepts. EATCS Monographs on Theoretical Computer Science (1992).
18. K.H.Kim: Approaches to Mechanization of the Conversation Scheme Based on Monitors. IEEE Transactions on Software Engineering SE-8 (1982) 189-197.
19. D.B.Lomet: Process Structuring, Synchronisation and Recovery using Atomic Actions. Proc. of ACM Conference Language Design for Reliable Software. SIGPLAN (1977) 128-137.
20. *PED*. `http://www-dssz.Informatik.TU-Cottbus.DE/~wwwdssz/` – the home page of PED (a Hierachical Petri Net Editor).
21. D. Peled: Combining Partial Order Reductions with On-the-fly Model-checking. Formal Methods in Systems design 8(1), 39-64 (1996).

22. *PEP.* `http://www.informatik.uni-hildesheim.de/~pep/HomePage.html` – the home page of PEP (a Programming Environment Based of Petri Nets).
23. M. Pezze, R.N. Taylor and M. Young: Graph Models for Reachability Analysis of Concurrent Programs. ACM Transactions on Software Engineering and Methodology 4/2 (April 1995) 171-213.
24. B. Randell: System Structure for Software Fault Tolerance. IEEE Trans. on Software Engineering 1(2) 220-232 (1975).
25. B. Randell, P. Lee and P. Treleaven: Reliability issues in computing systems design. ACM Computing Surveys 10(2): 123-165 (1978).
26. W.Reisig: Petri Nets. An Introduction. Springer-Verlag, EATCS Monographs on Theoretical Computer Science Vol.3, (1985).
27. S. Roch and P.H. Starke: INA: Integrated Net Analyzer, Version 2.2, Manual Humboldt-Univerität zu Berlin, Instutut für Informatik, April 1999.
28. S.M. Shatz, S. Tu, T. Murata and S. Duri: An Application of Petri Net Reduction for Ada Tasking Deadlock Analysis. IEEE Trans. on Parallel and Distributed Systems 7 (12), 1309-1324 (December 1996).
29. S.K.Shrivastava, G.N.Dixon and G.D.Parrington: An Overview of the Arjuna Distributed Programming System. IEEE Software 8 (1991) 66-73.
30. S.Tu, S.M.Shatz and T.Murata: Theory and Application of Petri Net Reduction for Ada-Tasking Feadlock Analysis. TR 91-15, EECS Dept., Univ. of Illinois, Chicago (1991).
31. A.Valmari: The State Explosion Problem. Lectures on Petri Nets II: Applications, Advances in Petri Nets. Lecture Notes in Computer Science 1492, Springer-Verlag (1998) 429-528.
32. A.J.Wellings and A.Burns: Implementing Atomic Actions in Ada 95, IEEE Transactions on Software Engineering 23 (1996) 107-123.
33. The Home page of Petri net Tools on the Web: `http://www.daimi.aau.dk/~petrinet/tools/`
34. The Home page of the Design/CPN tool: `http://www.daimi.au.dk/designCPN/`

# COALA: A Design Language for Reliable Distributed Systems Engineering

Julie Vachon[1], Nicolas Guelfi[2]

[1] Swiss Federal Institute of Technology,Programming Methods Laboratory,
1015 Lausanne Ecublens, Switzerland
email: Julie.Vachon@epfl.ch

[2] Luxembourg University of Applied Science, Software Engineering Competence Center,
L-1359 Luxembourg-Kirchberg, Luxembourg
email: Nicolas.Guelfi@ist.lu

**Abstract** This paper presents COALA, a language for the formal design of fault tolerant distributed systems. Advanced transaction models have already proved their interest for the design of reliable distributed systems. Unfortunately, these models often consist in low level algorithmic approaches which are not enough rigorously defined nor supported by a development methodology. In the field of formal approaches for distributed systems, we believe that effort is not sufficiently directed towards practical engineering where rigor is of interest. The principal contributions of this work are twofold: (1) it provides the Coordinated Atomic Action transaction model with a formal engineering framework; (2) it demonstrates the practical interest of the Petri net based object-oriented specification formalism CO-OPN/2 as an underlying tool allowing rigorous and methodological engineering of complex reliable distributed systems. This paper also shortly introduces a concrete example where high level Petri nets appear to be a very useful development tool when integrated in the software life cycle of complex distributed systems.

**Keywords**: distributed systems design, development methodology, formal methods, object orientation, Petri nets, fault tolerance, advanced transaction models.

## 1 Introduction

The concept of transaction has been developed to deal with the eventual loss of database integrity when concurrent programs operate on it. Indeed, these programs can be stopped due to hardware/software failures or can interfere with each other. In general, concurrency and failures are the two sources of potential errors that lead to database inconsistencies. For this reason, the transaction model is used as a basic unit of consistent and reliable computing within the database domain. Managing transactions is not an easy task and the subject has been studied thoroughly ([1], [17], [7]). Transaction management must deal with problems related to running sets of operations, that read or modify shared data, while always keeping the database in a consistent state, even when concurrent accesses and failures occur.

In order to increase the flexibility of the traditional transaction model some advanced models have been proposed. More flexibility is required particularly in contexts where long-lived and open-ended activities occur (such as in computer-aided design and manufacturing projects (CAD/CAM)). The ACID properties (atomicity, consistency, isolation, and durability) of the traditional model seem to strong and sometime inadequate for long duration activities which need to cooperate. Long lived activities are more inclined to conflicts since they often maintain locks for long period of time on numerous objects. Short transactions, waiting for lock resources, must be delayed. Long duration activities are also more vulnerable to failures. All these facts unfortunately tend to increases the risks for deadlocks and abortions. Hence, aborting a long transaction is clearly undesirable for a large quantity of important work might be lost. Moreover, the isolation property of the traditional model goes against the cooperation needs of these activities. The objectives of new models are of many kinds. Most of them try to statically divide or to dynamically restructure long transactions into smaller subtransactions so as to: increase concurrency and cooperation between transactions; relax the isolation property; and, in case of cancellation, propose alternative solutions other than coming back to the initial state.

Coordinated atomic actions (CA actions) present a general technique for achieving fault tolerance by integrating the concepts of conversation (that encloses cooperative activities), transaction (that ensures consistent access to shared objects), and exception handling (for error recovery) into a uniform structuring framework. More precisely, CA actions ([19],[20],[21],[23],[24]) use conversations as a mechanism for controlling concurrency and communication between threads that have been designed to cooperate with each other. Concurrent accesses to shared objects that are external to a CA action are controlled by the associated transaction mechanism that guarantees the ACID properties. In particular, objects which are external to a CA action, and can hence be shared with other concurrent CA actions, must be atomic and individually responsible for their own integrity. In a sense CA actions can be seen as a disciplined approach to using multi-threaded nested transaction while providing them with well-structured exception handling.

Although some efforts have been devoted to the formalization of advanced transaction models, this work clearly appears to be insufficient. To ensure the delivery of reliable complex distributed systems, it is important that formalization lies within the scope of a complete development process covering the whole software life cycle. This project therefore aims at providing the CA action model with a complete formal basis allowing system development according to a safe and methodological software engineering approach. To realize this, we first decided to provide the CA action model with a formal design language called COALA. COALA is meant for the definition of CA actions which are used to design complex distributed applications. COALA has a clear and simple syntax as well as a formal semantics which precisely defines the CA action model. COALA's formal semantics is expressed in the Petri net-based object oriented specification formalism CO-OPN/2 ([2],[3],[4]). CO-OPN/2 is a specification formalism which allows modular description, refinement, prototype implementation and testing. The main reason motivating the choice of CO-OPN/2 as the target language of COALA's semantics is that it includes most of the essential characteristics ([14]) necessary to provide formal, structured and operational semantics for distributed systems in which concurrency, atomicity and consistency of data structures are to be considered. A software engineering framework has been developed for CO-OPN/2 in order to provide methods and techniques supporting the main phases of the software life cycle (analysis, design, implementation, verification and validation). Even if all these techniques are not yet fully integrated in this project, they represent a good framework for the definition of a valuable development methodology for COALA. A major contribution of this paper is to show that high level Petri nets can provide useful semantic and methodological means for the development of high level transaction models like CA actions.

This paper is organised as follow; the second section presents the state of the art of new advanced transaction models for distributed systems, the third and fourth sections explain the syntax and semantics of COALA while the fifth section introduces a small banking example. The paper ends by shortly summarizing the development methodology we propose for COALA.

## 2 State of the Art

### 2.1 Traditional Transaction Systems

The main four properties associated with transactions are commonly known as the «ACID» properties.
(1) ***Atomicity*****:** a transaction is treated as a unit of operation, the effects of all the operations of a transaction persist or none do; (2) ***Consistency***: a transaction is a program which must map one consistent database to another; (3) ***Isolation***: concurrent transactions must be executed as if they were executed serially, in isolation (serializability), other transactions must not see the effects of any uncommitted transaction (failure isolation); (4) ***Durability***: the effects produced by a transaction must be made visible and permanent as soon as the transaction commits.

A transaction system satisfying these properties relieves the user from many hard problems related to the maintenance of the database consistency. Different algorithmic solutions exist which allow controlled access to shared data objects. The most well-known methods are the strict two-phase locking, the timestamp ordering and the optimistic concurrency control strategies. These methods differ in the strategy used to ensure serializability and recoverability of the concurrent transaction operations applied to shared data ([1],[7]).

## 2.2 Advanced Transaction Models

**Nested transactions** ([15]) are an intrinsically recursive model which provides for finer grained recovery and better control of reliable transaction execution. Appropriate design of subtransactions can help localize failures within a (parent) transaction. Parallelism and modularity can be exploited within transactions so as to enhance performance. The simplicity and the structuring capabilities of this model help managing the complexity of systems. On the other hand, the nested transaction model remains quite conservative regarding solutions proposed to control concurrency and to maintain data consistency.

The **split-transaction model** ([16]) was proposed as an answer to some specific transaction problems encountered in open-ended activities. The main objective is to allow dynamic reconfiguration of long transactions and to restructure them so as to reflect some possible dynamic change in the (user or application) requirements. In the split-transaction model, partial results (that will not change) can be committed, thus preventing or reducing loss of work in case of failure. This early releasing of committed resources allows more concurrency while reducing isolation between transactions and still ensuring serializable accesses to resources for all transactions. However, for non-interactive applications, the usefulness of this model is less obvious.

The **saga model** ([13]) relaxes the requirement that a long transaction be executed as a *single* atomic action. Since shorter transactions reduce probabilities of conflicts, failures, deadlocks and abortions, a saga is defined as a set of component subtransactions executed in sequence or in parallel. Isolation being limited to subtransactions (not to the whole saga), this model allows more flexible cooperation between long activities represented by sagas. Moreover, the concept being relatively simple, the saga model does not require very complex or novel implementation mechanisms. Contrarily to nested transactions, this model allows only two levels of nesting and the outer level (saga) doesn't provide full atomicity. Sagas must be designed statically and do not allow dynamic restructuring. There is no high-level support (loop, branch, etc.) to control the execution flow of component subtransactions within a saga, neither are there mechanisms allowing conditional creation of component subtransactions. As for recovery, the model doesn't deal with the failure of compensating transactions and hence does not guarantee their successful completion.

The **flex transaction model** ([12]) solves transaction processing problems involving data in multiple autonomous an possibly heterogeneous database systems. Transactions accessing multiple database are often long-lived and may need to cooperate. Proposed features contribute to a more flexible and precise ordering of subtransactions execution. However, with regards to sagas, the Flex model neither really increases cooperation nor ameliorate consistency maintenance. Unfortunately, as for sagas, structuring possibilities remain limited to a two-level nesting, while failure and concurrency atomicity are still not ensured at the outmost level.

The **ConTract model** ([22]) proposes mechanisms to facilitate persistence, consistency, recovery, semantic synchronization and cooperation. Failure and concurrency atomicity is not ensured at the ConTract level but the model proposes some alternative solutions more adapted to the requirements of long lived activities: recovery mechanisms based on context management (saving/restoring), consistency control using compensation and semantic synchronization, etc. Contrarily to former models presented in this section, inconsistency problems due to the relaxation of the isolation property have been addressed. However, the ConTract model still leaves place for inconsistency, but this seems the price to pay when isolation isn't guaranteed any more and when actions need to be compensated instead of being aborted.

## 2.3 Discussion: CA Actions and Advanced Transaction Models

CA actions exhibit all the ACID properties and thus ensure the maintenance of object consistency. As for nested transactions, CA actions provide modularity, safe concurrency and finer grained recovery. Since CA actions are multi-threaded units of work, cooperation is encapsulated inside the action: threads cooperate inside a CA action by exchanging information through internal objects. There is therefore no need to break or relax the isolation property to allow for more cooperation. Cooperation is made easy and safe. Contrarily to most of the models introduced, control flow between subtransactions can be specified: inside a CA action, threads execute sequential instructions and can thus control the execution of subtransactions using sequence and conditional instructions for example. Long transactions can easily be designed and divided into subtransactions so as to release resources earlier for other competing subtransactions and to allow for finer grained recovery in case of failure. The CA action model also provides means for coordinated forward error recovery, thus allowing alternative solutions to be executed (by the participating threads of a CA action) instead of aborting and coming back to the initial state. The model also allows for persistency issues and recovery from system failures. Finally, and similarly to the ConTract model, CA actions must not be considered as yet another transaction model but as a new comprehensive framework which aims at providing all the adequate mechanisms necessary to ensure both the modular design and the safe execution of large reliable distributed systems. CA action concepts are introduced in Section 3 through the presentation of COALA.

## 2.4 Formalization and Methodology

Reliable distributed systems development is a complex and costly activity. We believe that a formal and methodological framework can be of great use for engineers.

Formalization is an important step in language and model design which contributes to clarify concepts, syntax and semantics; it can help classifying models and facilitates formal reasoning. Transaction systems presented in the previous section have more or less been formalized. Nested transactions and split-transactions have shortly been specified in ACTA ([5]). Atomic actions and sagas have also been characterized using ACTA in [6], which allowed better comparison of some of these models' properties. As for ConTracts, mechanisms to describe and execute their computations have been introduced in [18]. Finally, Flex transactions have been formalized using Predicate Petri Nets ([12]).

As far as we know, formal development methodologies for advanced transaction models have not been deeply investigated and work still needs to be done in this area. Most researchers who have taken up the subject of advanced transaction models have looked at it from an algorithmic and implementation point of view. Our work rather positions itself at a higher level, trying to integrate the CA action transaction model into a formal and methodological engineering framework. We believe, as shown in this paper, that high level Petri nets and more precisely the CO-OPN/2 formalism, are a ideal tool for building this framework.

## 3 The COALA Language

### 3.1 COALA Basic Concepts

This subsection shortly explains the main concepts of CA actions ([23],[24]) as they are considered in COALA ([19],[21]).

**Roles.** A CA action is viewed as a collection of roles, each of them being associated with a portion of code (a subprogram). The ultimate goal of a CA action consists in coordinating its roles so as to coherently manage all the system's software entities, i.e. the system's objects. An execution thread enters a CA action by activating the role it wants to execute. When each role has been activated, the CA action starts and each thread thus executes its role. This execution is coordinated by the CA action which sees that all ACID properties (which ensure the consistent state of objects) are respected.

**Objects.** The CA action concept defines two kinds of objects, namely *internal* and *external* objects. An internal object is an object local to a CA action and it is shared between all its roles. It is used for the coordination of roles or for other internal computations. An external object is a global object which can be modified by the roles of different CA actions. Operations on external objects are constrained by the ACID properties. External objects may be shared simultaneously by several CA actions but the effect of the operations applied to them must be the same as if the CA actions had been executed one after another. In other words, the schedule of operations applied to external objects must be *serializable*. References on external objects are passed to roles through their parameters. Internal objects are declared inside the CA action's body.

**Outcomes.** The execution of a CA action consists in coordinating the execution of the its roles. A CA action ends with one of the following outcomes:
- *Normal outcome*. The ACID properties were satisfied during execution and the CA action commits its operations;
- *Exceptional outcome*. The ACID properties were satisfied but an exception must be signalled to the enclosing CA action;
- *Abort outcome*. The CA action has aborted and has undone its operations while satisfying the ACID properties;
- *Failure outcome*. A major problem occurred, preventing the CA action not only from committing but also from aborting (i.e. the CA action ends without guaranteeing the satisfaction of the ACID properties).

All the roles of the CA action end with the s*ame* outcome; in the case of an *exceptional outcome* all the roles signal the s*ame* exception to the enclosing CA action.

**Exceptions and Handlers**. There are two types of exceptions in CA actions: *internal* and *interface* exceptions. Internal exceptions are local to a given CA action, which must therefore handle them on its own; each role of a CA action has a set of subprograms, called handlers, to handle these internal exceptions or a combination of them. Internal exceptions are *raised* by roles. When some roles of a CA action *raise* different internal exceptions concurrently, all roles are interrupted and a resolution algorithm then determines which handler must be activated by all the roles to handle these concurrent exceptions. Handlers are subprograms which try to bring the system into a new consistent state. Handlers are not authorized to *raise* internal exceptions during their execution. As for interface exceptions, they are *signalled.* When a role *signals* an exception, it stops its current execution, waits for the other roles to end and agrees with them to propagate the exception at the outside level, that is to say to the enclosing CA action. Both roles and handlers can *signal* interface exceptions. When signalled exceptions are propagated, the situation is the same as if these exceptions were raised at the level of the enclosing CA action. In addition, two default interface exceptions are made available to all CA actions: the Abort exception and the Fail exception. These exceptions can be *raised* or *signalled*.

**Behaviour of CA actions.** The execution of a CA action always corresponds to one of the following scenario:
**(1)** Each role executes and ends successfully. No exception is raised during execution of roles. The CA action ends with a normal outcome;
**(2)** Some of the roles concurrently *raise* internal exceptions during the execution. These raised exceptions can be identical or different. In any case, the CA action uses a resolution graph to decide which exception handler has to be executed to cope with these concurrent exceptions. All the roles of the CA action are informed of the exceptions being raised and of the exception handler which they must execute. Once handlers are activated, the following cases can occur:

* If all the exception handlers end successfully, the CA action ends with a normal outcome;

* If some of the exception handlers *signal* an interface exception during their execution and if these exceptions are the same, then the underlying system forces all the handlers to signal this exception to the enclosing CA action. The CA action is said to end with an exceptional outcome;

* If some of the exception handlers *signal* an interface exception during their execution but these exception are different, the underlying system tries to abort the CA action. If it succeeds, the CA action ends with the abort outcome and all the roles *signal* an Abort exception to the enclosing CA action; if the abort operation fails, the CA action ends with the failure outcome and a Fail exception is *signalled* to the enclosing CA action. The underlying system also tries to abort the CA action if an exception is *raised* in a handler program. For example, this case can occur when a nested CA action, called during the execution of a handler, *signals* an exception which is consequently *raised* at the handler level. Since *raised* exceptions are not allowed in handlers, the underlying system must abort the action.

**(3)** Some of the roles *signal* an interface exception during their execution and these exceptions are all the same. In this case, the underlying system forces all the roles to *signal* this exception to the enclosing CA action. The CA action ends with an exceptional outcome;
**(4)** Some of the roles *signal* an interface exception during their execution but these exceptions are different. In this case, the underlying system performs an abort operation and if it succeeds all the roles *signal* an Abort exception and end with an abort outcome; if it fails, they all *signal* a Fail exception and end with an failure outcome;
**(5)** A role *signals* an interface exception while another role *raises* an internal exception. The raised exception is ignored, and the CA action continues executing accordingly to cases (3) or (4).

### 3.2 COALA Syntax

COALA has a simple and clear syntax to define CA actions and their components (exceptions, roles, handlers, etc.) However, for the definition of objects and expressions, COALA uses the CO-OPN/2 formal specification language. Each COALA program thus comprises a set of CO-OPN/2 modules which define abstract data types and object classes. We do not present CO-OPN/2's syntax and in this part, but it can be found in ([2]).

**Interface and body.** The COALA definition of a CA action is made of two parts: an Interface section, which is visible to other CA actions, and a Body section, which is private and hidden to the outside world. The Interface of a CA action has the following shape:

```
CAA <CAAName>
Interface
    Use                              // The list of CO-OPN/2 modules (abstract
        <ModuleName1>,               // data types and classes) which are used
        <ModuleName2,>, ... ;  // in the Interface part of the CA action.
    Roles
    // The list of its (parameterized) roles
        <RoleName1> : <Parameter type>, ...;
        <RoleName2> : <Parameter type>, ...; ...
    Exceptions
    // The list of the (parameterized) interface exceptions which the CA action can
    // signal to an enclosing CA action
        <ExceptionName> : <Parameter type>;          ... ;
Body
    Use                              // The list of CO-OPN/2 modules (abstract data
        <ModuleName1>,               // types and classes) which are used in
        <ModuleName2>, ...;   // the Body part of the CA action.
    Use CAA          // The list of nested CA actions.
        <CaaName1>, <CaaName2,>, ... ;
Objects
    // The list of the CA action's internal objects; the behaviour of
    // these objects are specified in separate CO-OPN/2 specification modules.
    <Object1>:<type>;
    <Object2>:<type>; ...;
    Exceptions                                          // The list of the (parameterized)
        <ExceptionName> : <Parameter type>; // internal exceptions that can be
                                                        // raised within the CA action.
    Handlers
    // The list of the (parameterized) exception handlers of the CA action.
    <HandlerName1> : <Parameter type>, ...;
    <HandlerName2> : <Parameter type>, ...;
    Resolution
    <ExceptionName1>(<ParameterName,...>),<ExceptionName2>(<ParameterName,...>)
                -> <HandlerName>(<ParameterName,...>);
        // The CA action resolution graph which lists all the combinations
        // of internal exceptions which can be raised concurrently, together with the
        // handlers which must be activated in each case. If a combination of internal
        // exceptions can't be found in the list during execution, this means that no handler //
        was foreseen to handle the case: an abort exception must be signalled by the
        // underlying system.

    Where   <VarName1>: <TypeName1>;  // Local variables and their types;
            <VarName2>: <TypeName2>;       // Types are specified using CO-OPN/2.»
            ...
    Role <RoleName1> (<ParameterName>, <ParameterName>, ...);
        Begin    <instructions>;End                // Role program.
        Where <VarName>: <TypeName>;  // Local variables and their types;
                ...
        Handler <HandlerName>;
            Begin <instructions>; End    // Role program.
            Where <VarName>: <TypeName>; // Local variables and their types;
                    ...
        End <HandlerName>;
    End <RoleName1>;

    Role <RoleName2> (<ParameterName>, ParameterName>, ...);
    ....
End <CAAName>;
```

**Roles and handlers instructions.** The behaviour of a role or a handler is described by an *instruction block*, that is a sequence of instructions. COALA's instructions may contain variables but also expressions and conditions which refer to CO-OPN/2 specification modules declared in the Use fields of a CA action.

- ***Variable.*** Name to which a (typed) value is associated. A variable must be in the scope of the *instruction block* where it is used. Variables are declared in the Where field of a role, a handler, a resolution graph, etc.
- ***Expression***. Since COALA uses CO-OPN/2 specifications for the definition of data types and object classes, an expression is a term written over the global signature of a CO-OPN/2 specification and over some sorted set of declared COALA variables.
- ***Condition***. Boolean expression based on the terms built over the CO-OPN/2 global signature.
- ***Instructions*** An *instruction block* is a non empty sequence of *instructions* delimited by Begin and End keywords. An instruction is either empty or is one of the following:
    1. **Assign** a **To** v; - Assigns expression a to variable v.
    2. **Execute** o.m(a1,..., an); - If method m of the object referenced by o can be exe-

cuted given parameters a1,..., an and according to the corresponding CO-OPN/2 specification, then the instruction succeeds. If not, then an internal exception (i.e. a predefined exception raised by the underlying system) is raised.

3. **If** c **Then** instructionBlock1 **Else** instructionBlock2; - If condition c is true (according to the model of the given CO-OPN/2 specification), then instructionBlock1 is executed. If c is false, then instructionBlock2 is executed.
4. **Raise** exceptionName(a1,..., an); - Allows a role to *raise* an internal exception within a CA action. exceptionName must be the name of an internal exception defined in the body of the CA action or the Abort or Fail exceptions. a1,..., an are expressions which parameterize the exception raised.
5. **Signal** exceptionName(a1,..., an) - Allows a role or an exception handler to *signal* an interface exception to an enclosing CA action. The name of the exception (exceptionName) is either one of the exceptions defined in the interface of the CA action, or the Abort or Fail exception. When signalling an exception, a role/handler interrupts its execution and waits for the other roles/handlers of the CA action to end so as to propagate the exception being signalled.
6. **Call** roleName(a1,..., an) **Of** caaName - Allows the activation of role roleName of (nested) CA action caaName. If an instance of caaName already exists and its role roleName hasn't yet been fulfilled, then roleName is activated with its actual parameters a1,..., an passed on. Otherwise, the underlying system first creates a new instance of caaName, (with its own internal objects), before activating the role roleName. The actual parameters a1,..., an are expressions which may contain variables referring to external objects. Note that this instruction is blocking and execution resumes only after CA action caaName is ended.

### 3.3 COALA Semantics

COALA's semantics ([20],[21]) is defined as a translation from COALA programs into their formal description in CO-OPN/2. Since objects and abstract data types of a COALA programs are already defined in CO-OPN/2, part of the semantics is directly obtained. The other part must deal with the semantic definition of CA actions coordination mechanisms, of roles/handlers programs and of external objects atomicity. For this, we propose a set of CO-OPN/2 generic classes specifying the general behaviour of (1) CA actions, (2) Roles and (3) Internal/External objects affected by CA actions. Each CA action, each role and each object of a COALA program is being semantically described by a CO-OPN/2 object, which is an instance of one of the three generic classes or is an instance of one of their subclasses. Indeed, particular instances and subclasses must be created to express the precise semantics of a given COALA program.

- **Class** Caa describes all the coordination mechanisms of the CA action. This includes starting all the roles synchronously, solving distributed raised exceptions and managing how they must be handled, coordinating ending roles, signalling exceptions, etc.
- **Class** Role describes how a role is started, how it ends and how it interprets the instructions which it has to execute (see instructions described in Section 3.2)
- **Class** IntExtObject describes how an object in a CA action processes the operation

requests it receives, while together ensuring respect of the operations' ACID properties. Class IntExtObject refers to another CO-OPN/2 class called objVersion to keep trace of the different object state versions that each CA action sees at one given time. More precisely, these object state versions represent the different views that each individual CA action (operating on the object) has of the object's state. When a CA action commits, the view it has of the object's state becomes the current state of the object. If it aborts, no change is made to the object's current state.

This object-oriented approach adopted to build COALA's semantics presents many advantages. Among others, this semantic definition is modular, generic, clear and quite intuitive. The semantics of a COALA program is obtained by extending the generic classes introduced above and by creating the appropriate instances which represent the CA actions in the program and their components (roles, internal/external objects, etc.) Figure 1 illustrates an example of a CO-OPN/2 class hierarchy used for the semantics of a COALA program. The genericity and modularity qualities of this approach allow to translate COALA programs into a set of CO-OPN/2 objects by a simple extension of the generic classes (Caa, Role, IntExtObject) into specific subclasses and by creating appropriate instances of these subclasses.

We give below a short and partial illustration of how roles are translated into CO-OPN/2 objects belonging to the abstract class Role which specifies the generic mechanisms and the basic behaviour that any role must have. Roles specific to a CA action are described by individual classes which inherit from the abstract class Role. Each of these subclasses allow the creation of role objects having a particular program to execute.
A role object is a CO-OPN/2 object which is itself a high-level Petri net. As shown on Figures 2 and 3, a role object is composed of

- a set of internal transitions (white rectangular boxes) which specify its internal behaviour. The main internal behaviour of a role object consists in interpreting the instructions which the role has to execute.
- a set of methods (black rectangular boxes) which allow other CO-OPN/2 objects to modify its state. In the semantics Caa objects uses role object methods to act on them and coordinate their execution (to interrupt roles, to synchronize them, etc.)

As mentioned, the main internal transition of role objects is called eval and is used for interpreting COALA instructions. Figure 2 illustrates the evaluation of the instruction "**Call** roleName(a_1,...,a_n) **Of** caaName". As shown on the picture, the instruction is read from place Instr. To interpret it, the evaluation context (containing the values assigned to variables) is required and is thus taken from place Ctxt and put back into it right after. When evaluating a Call instruction, transition eval puts the identity of the role to be called along with its evaluated parameters in place RoleToCall. These values will be fetch from this place as soon as the CA action object whose role is being called can synchronize with the method callRole of the calling object. This synchronization is illustrated by Figure 3. On this figure, the Call instruction is performed by role object *c* while the role object being called is identified by *r*. The interpretation of this instruction consists in evaluating the arguments in the context and then putting the result together with the identity of the role object to be called (i.e. *r*) in place RoleToCall. Appropriate tokens in this place allow method callRole of role object *c* to eventually fire. It does indeed as soon as the method startRoles of the caa object coordinating role *r*, tries to synchronize sequentially with (1) the method callRole of object *c* and (2) the method start of object *r*.

Role objects have many other methods (not shown on Figures 2 and 3) which are used to define the exception handling mechanism, the way roles operate on CA actions external objects, etc.

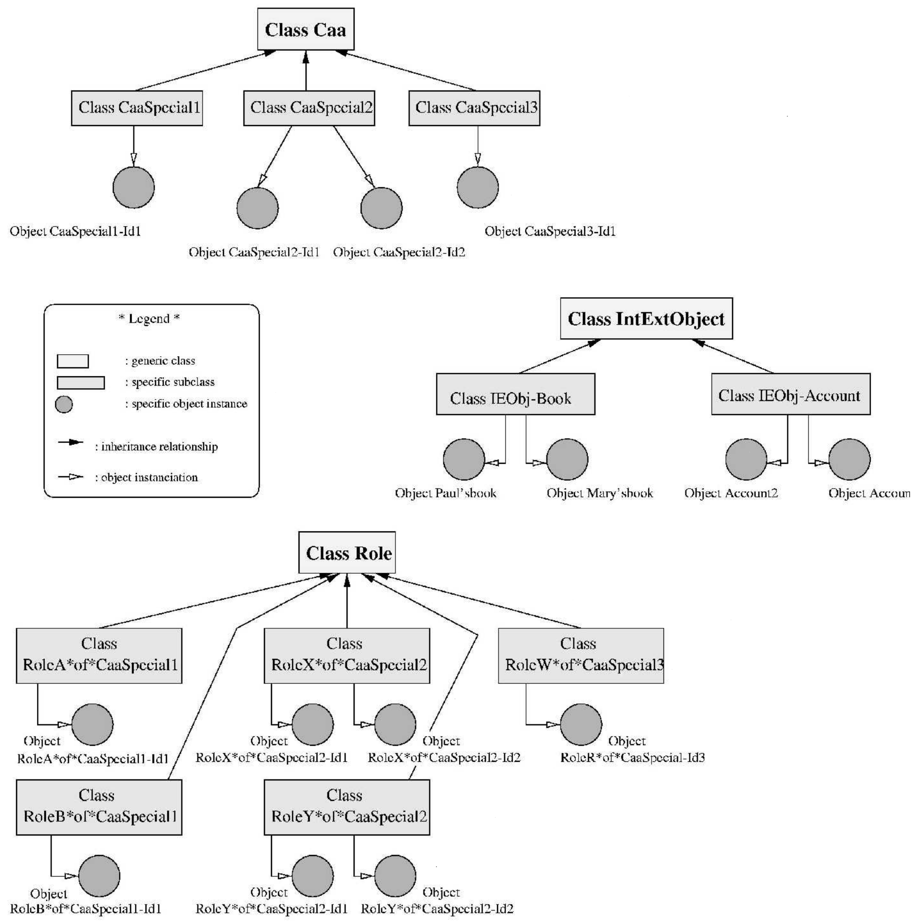

Class Caa
Class CaaSpecial1
Class CaaSpecial2
Class CaaSpecial3
Object CaaSpecial1-Id1
Object CaaSpecial2-Id1
Object CaaSpecial2-Id2
Object CaaSpecial3-Id1
* Legend *
: generic class
: specific subclass
: specific object instance
: inheritance relationship
: object instanciation
Class IntExtObject
Class IEObj-Book
Class IEObj-Account
Object Paul'sbook
Object Mary'sbook
Object Account2
Object Accoun
Class Role
Class RoleA*of*CaaSpecial1
Class RoleX*of*CaaSpecial2
Class RoleW*of*CaaSpecial3
Object RoleA*of*CaaSpecial1-Id1
Object RoleX*of*CaaSpecial2-Id1
Object RoleX*of*CaaSpecial2-Id2
Object RoleR*of*CaaSpecial-Id3
Class RoleB*of*CaaSpecial1
Class RoleY*of*CaaSpecial2
Object RoleB*of*CaaSpecial1-Id1
Object RoleY*of*CaaSpecial2-Id1
Object RoleY*of*CaaSpecial2-Id2

**Figure 1 :** Inheritance hierarchy and object instances for the semantics

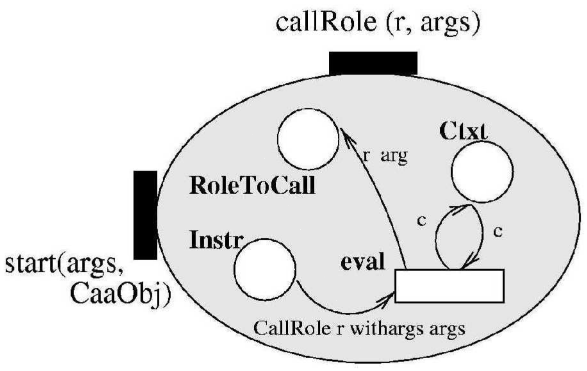


**Figure 2 :** Role object c eva1uating a CallRole instruction

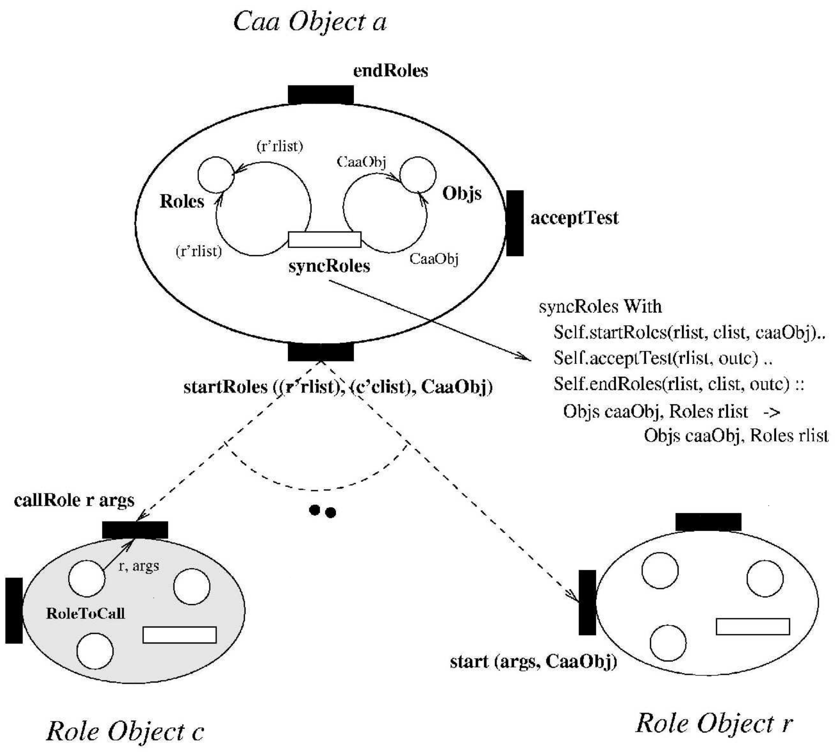


**Figure 3 :** Synchronisation between the calling role object c and the called role object r

## 4 Small Example

This section introduces a small banking example to illustrate COALA concepts and syntax. The complete COALA design of this example is given in [19]. Of course, this simple example can not fully illustrate the power of COALA and its interest for reliable distributed systems engineering. For this purpose, [21] presents a complete case study where an Internet auction service is designed using COALA and is implemented using a Java library especially designed for COALA. The example considered in this section illustrates a joint withdraw implicating two persons and two joint accounts. These two persons own together two joint accounts and want to withdraw a certain amount of money from the joint account having the highest balance.

The bank with which these two persons deal, offers its clients a special kind of account called "joint account". A joint account is owned by two clients, called co-owners. Each owner is given a personal identification number (pin) which he must use to identify himself. The conditions applied to joint accounts and pins are the following:

- A withdraw operation on a joint account requires the authorisation of both its co-owners.
- The balance of the joint account can be consulted by any co-owner (without the authorisation of the other co-owner).
- Money can be deposited (by anyone) on the joint account without any authorisation.
- A client gives his authorisation for the execution of a transaction by providing his pin, which must henceforth be validated. If he makes a mistake, validation fails and the operation is immediately aborted.

The COALA program corresponding to this banking example is made of two parts:

- a COALA specification consisting of two main CA actions: JointWithdraw and Withdraw. These CA actions refer to three other small complementary CA actions (WaitPIN, WaitInfoAmount and WaitReadAmount) which are simply used to force the synchronization of threads.
- a CO-OPN/2 specification (classes and data types) describing the objects used by the CA actions: IntegerContainer, PIN and Account.

**CA action JointWithdraw.** The JointWithdraw CA action presents two clients, named client1 and client2. They are the co-owners of two joint accounts, account1 and account2. Each joint account has an appointed co-owner who takes care of the main transactions operated on the account: client1 is responsible for transactions on account1 while client2 is responsible for transactions on account2.

In CA action JointWithdraw, role client1 describes the behaviour of a client who wants to withdraw a certain amount of money out of one of two given joint accounts. More precisely, the money must be withdrawn from the account having the greatest balance. Client1 thus informs the other co-owner, client2, about the amount to withdraw. Each client consults his appointed account and tells the other how much money there is left. Each client henceforth knows on which account the money must be taken from. The client responsible for the account having the highest balance must perform the withdraw by calling role withdrawer of CA action Withdraw. On his side, the other client calls role partner of this same CA action.

However, if there is not enough money on a single account, the missing amount is drawn out the other account. If some money is still missing, the exception NotEnoughMoney must be signalled. Figure 6 partially describes the COALA design of CA action JointWithdraw. Figure 4 illustrates one of the normal (i.e. with no exception) execution of CA action JointWithdraw. Boxes delimit the execution part under the control of each CA action. Circles represent objects. As for "X" symbols, they denote Execute instructions in the role programs.

**CA action Withdraw.** As mentioned, a withdraw operation requires the authorisation of both co-owners. Hence, the client withdrawing the money (the withdrawer) must not only give is own pin but must also get the pin of the other co-owner (his partner). Note that clients have a single try to enter their pin when they are prompted for it. If the wrong pin is entered, then an Abort exception is raised, and the whole CA action Withdraw is aborted.
If the pin validation process succeeds, two cases can occur:

- There is enough money on the account. The required money is drawn out the account and the balance is updated.
- There is not enough money on the account. All the money on the account is drawn out, the balance is thus put to 0 and the exception missing is raised. The withdrawer's exception handler takes up the execution and indicates the amount of missing money to the enclosing CA actions by assigning this amount to the external object commonAmount. If no problem occurs during this handling phase, CA action Withdraw simply ends with a normal outcome.

Figure 5 illustrates the normal execution of CA action Withdraw.

**CA actions WaitPin, WaitInfoAmount and WaitReadAmount.** These nested CA actions are used to coordinate the work of two threads, more precisely to synchronize these two threads. They all have the same shape and achieve the same work: a first thread enters the CA action by calling one of the two roles; this thread is suspended until the second role is called by another thread. Since the body of the roles are empty, these CA actions end (with a normal outcome) right after the initial synchronization of the roles.

.

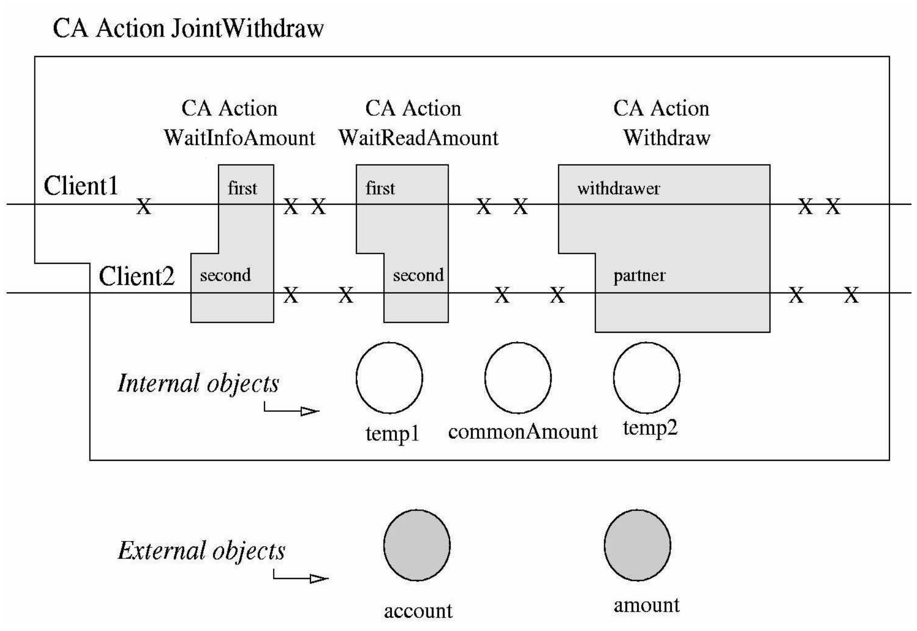


**Figure 4 :** A normal execution of CA action JointWithdraw

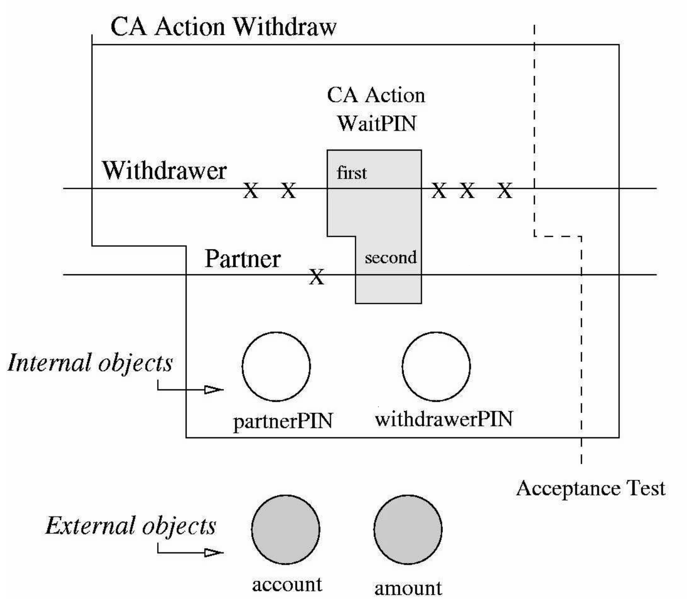


**Figure 5 :** A normal execution of CA action Withdraw

```
CAA JointWithdraw;
Interface
Use   Account, Integers;
Roles
        client1 : accountType, integer;
        client2 : accountType;
Exceptions
        NotEnoughMoney;
Body
Use Booleans;
Use CAA Withdraw, WaitInfoAmount, WaitReadAmount;
Object temp1,temp2,commonAmount: integerContainer;
Handler
        FailHandler; Aborthandler;
Resolution
        Abort               -> Aborthandler;

Role client1 (account, amount);
        Begin
         Execute commonAmount.put(amount);
         Call first of WaitInfoAmount;
         Execute account.balance(money);
         Execute temp1.put(money);
         Call first of WaitReadAmount;
         Execute temp1.get(t1);
         Execute temp2.get(t2);
         If ((t1>=t2)=true) then Begin
                 Call withdrawer(account,
                                                  commonAmount)
                        of Withdraw;
                 Execute commonAmount.get(ca);
                 If (ca > 0 = true) Then
                        Call partner of Withdraw;
         End
         Else Begin
                Call partner of Withdraw end;
                Execute commonAmount.get(ca);
                If (ca > 0 = true) Then
                        Call withdrawer(account,
                                             commonAmount)
                                of Withdraw;
         End;
         Execute commonAmount.get(ca);
         If (ca > 0 = true) Then
                signal NotEnoughMoney;
        End

Where t1, t2, ca, money, amount: integer;
         account: accountType;
        Handler FailHandler;
        Begin Signal Fail; End
        End FailHandler;

        Handler Aborthandler;
        Begin Signal Abort; End
        End Aborthandler;
End client1;

Role client2 (account);
        ...
```

*// The instruction block of this role is very similar to the one found in role client1, it follows a symmetric pattern. Among others, client2 enters nested CA actions as client1 does, following the same if-then-else scheme, but calling the opposite roles. Another difference is that client2 has only one parameter: no amount parameter. Indeed it is only client1's responsibility to indicate how much money must be withdrawn. Handlers are identical for both roles. //*

```
End client2;
End JointWithdraw;
```

*Comments:*

*(1) In CA action JointWithdraw, temp1, temp2 and commonAmount are local objects allowing roles client1 and client2 to communicate between each others. These objects behave like buffers and thus provide methods such as get and put to modify and access their content.*

*(2) CA actions WaitInfoAmount and WaitReadAmount are used to synchronize client1 and client2 so that they can get values of local objects at the timely moment.*

*(3) The commonAmount object contains the amount of money that must be withdrawn. It is passed as an external parameter to CA action Withdraw which decrements its value accordingly.*

**Figure 6 :** COALA design of CA action JointWithdraw

## 5 Development Methodology

COALA is a formal language for the design of distributed systems; it has a syntax, a semantics but no development methodology yet. One of the objectives of this work aims at providing COALA with the development methodology developed for CO-OPN/2. CO-OPN/2's development methodology is presented in details in [8], [9] and its application is shown in [10] and [11]. It provides an integrated formal framework designed especially for this class of high level Petri nets which covers specification, design, implementation, simulation and test. The work presented in this paper aims at adapting this engineering framework to COALA. The development methodology for COALA will mainly consists in the following phases:

- Definition of requirements (functional and safety requirements).
- Abstract design in COALA.
- Formal refinement of the design consisting in several steps and using an adapted notion of a refinement function for COALA to progressively reach a detailed design.
- Implementation carried out as a direct translation of the detailed design into an implementation language (eventually using a set of predefined libraries)
- Test, using test sets allowing for the formal CO-OPN/2 semantics of the design written in COALA.
- Simulation, automatic generation of executable code based on the axiomatic semantics of CO-OPN/2.

Validation and verification could be partly addressed, in the COALA development methodology, using the CO-OPN/2 notion of "contract". A contract is assigned to each COALA design step (abstract or refined). Contracts must be preserved during the whole development process. Hence, properties expressed and satisfied at the abstract level will still be present at implementation level. Formal and informal proofs are proposed to guarantee the preservation of this property. Contracts are means to address validation and verification issues.

## 6 Conclusions

This paper had two main objectives. First, we introduced COALA as a formal language for the design of reliable distributed systems based on the concepts provided by the Co-ordinated Atomic Action framework. Secondly, we have described the interest of the formal engineering framework of CO-OPN/2 to address semantic and methodological issues in the context of distributed systems development. This was made by defining a new design language, COALA, with a precise syntax and a semantics given in terms of CO-OPN/2 specifications. We have then summarized how techniques available in the CO-OPN/2 framework could be exploited to provide COALA with an engineering methodology allowing formal development of distributed systems. This paper represents a contribution which aims at increasing the level of rigor and methodology used in the engineering of distributed systems designed using an advanced transaction scheme based on high level Petri nets.

## 7 Acknowledgments

This work has been partially supported by the Esprit Long Term Research Project 20072 “Design for Validation” (DeVa) with the financial support of the OFES (Office Fédéral pour l'Education et la Science), and by the Swiss National Science Foundation'. We would like to thank our colleagues for all the helpful discussions and collaboration we had concerning this work.

## References

[1] P.A. Bernstein, V. Hadzilacos and N. Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Series in Computer Science. Addision-Wesley, 1987.

[2] O. Biberstein. *CO-OPN/2: An Object-Oriented Formalism for the Specification of Concurrent Systems*. PhD thesis, University of Geneva, Geneva, Switzerland, 1997. Thesis No 2919.

[3] O. Biberstein, D. Buchs and N. Guelfi. *Object-Oriented Nets with Algebraic Specifications: The CO-OPN/2 formalism*. InAdvances in Petri Nets on Object-Orientation, G. Agha and F. De Cindio (Ed.), Lecture Notes in Computer Science, Springer-Verlag, 2000. *To appear*.

[4] D. Buchs and N. Guelfi. A concurrent object-oriented Petri nets approach for system specification. In M. Silva, editor, *12th International Conference on Application and Theory of Petri Nets*, pages 432–454, Aarhus, Denmark, June 1991.

[5] P.K. Chrysanthis and K. Ramamritham. Acta: a framework for specifying and reasoning about transaction structure and behavior. In *Proceedings of the ACM Special Interest Group on Management of Data* (SIGMOD) 1990, pp 194-203, New York, 1990. ACM Press.

[6] P.K. Chrysanthis and K. Ramamritham. Acta: The saga continues. In *Database Transaction Models for Advance Applications*, chapter 10, p. 349-397, Morgan Kaufmann Publishers, 1992.

[7] G. Coulouris, J. Dollimore and T. Kindberg. *Distributed Systems: Concepts and Design*. Addison-Wesley, second edition, 1994.

[8] G. Di Marzo Serugendo. A Formal Developement and Validation Methodology for System Design. In *Fifth International Conference on Information Systems Analysis and Synthesis* (ISAS'99), 1999.

[9] G. Di Marzo Serugendo. *Stepwise Refinement of Formal Specifications Based on Logical Formulae: from COOPN/2 Specifications to Java Programs*. PhD thesis, Swiss Federal Institute of Technology (EPFL), Lausanne, Switzerland, 1999. Thesis No 2919.

[10] G. Di Marzo Serugendo and N. Guelfi. Using Object-Oriented Algebraic Nets for the Reverse Engineering of Java Programs: A Case Study. In *Proceedings of the International Conference on Application of Concurrency to System Design* (CSD'98), IEEE Computer Society Press, 1998, pp. 166-176. Also available as Technical Report (EPFL-DI No 98/267).

[11] G. Di Marzo Serugendo, N. Guelfi, A. Romanovsky and A. F. Zorzo. Formal Development and Validation of Java Dependable Distributed Systems. In *Fifth IEEE Internation-*

*al Conference on Engineering of Complex Computer Systems* (ICECCS'99), IEEE Computer Society Press, 1999.

[12] A.K. Elmagarmid, Y. Leu, W. Litwin and M. Rusinkiewicz. A multidatabase transaction model for interbase. In *Proceedings of the 16th VLDB Conference*, pages 507-518, Brisbane, Australia, 1990.

[13] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of the ACM Special Interest Group on Management of Data* (SIGMOD) 1987, pages 249-259, San Francisco, may 1987.

[14] N. Guelfi, O. Biberstein, D. Buchs, E. Canver, M-C. Gaudel, F. von Henke and D. Schwier, *Comparison of Object-Oriented Formal Methods*, Technical Report of the Esprit Long Term Research Project 20072 ‘‘Design For Validation’’, University of Newcastle Upon Tyne, Department of Computing Science, 1997.

[15] J.E.B. Moss. *Nested Transactions: An introduction*, chapter 14, pages 395-425. Van Nostrand Reinhold, New York, 1987.

[16] C. Pu, G. Kaiser and N. Hutchinson. Split-transactions for open-ended activities. In *Proceedings of the 14th international conference on VLDB*, pages 26-37, Los Angeles, September 1988.

[17] K. Ramamritham and P.K. Chrysanthis. *Advances in concurrency control and transaction processing*. Executive Briefing Serie. IEEE Computer Society Press, 1997.

[18] F. Schwenkreis. A formal approach to synchronize long-lived computations. In *Proceedings of the 5th Australasian Conference on Information Systems*, Melbourne, 1994.

[19] J. Vachon, D. Buchs, M. Buffo, G. Di Marzo Serugendo, B. Randell, A. Romanovsky, R.J. Stroud and J. Xu. Coala - a formal language for coordinated atomic actions. In *3rd Year Report, ESPRIT Long Term Reaseach Project 20072 on Design for Validation*. LAAS Francd, november 1998.

[20] J. Vachon, *The Semantics of COALA in CO-OPN/2*, Technical Report EPFL-DI No 98/300, Swiss Federal Institute of Technology in Lausanne, CH-1015, Lausanne, Switzerland, June 1999.

[21] J. Vachon, *COALA: A design language for reliable distributed systems*, PhD thesis, Swiss Federal Institute of Technology (EPFL), Lausanne, Switzerland, 2000. *To appear*.

[22] H. Wachter and A. Reuter. The contract model. In *Database Transaction Models for Advance Applications*, chapter 7, pages 219-263. Morgan Kaufmann Publishers, 1992.

[23] J. Xu, B. Randell, A. Romanovsky, R.J. Stroud, A.F. Zorzo, E. Canver and F. von Henke, Rigorous Development of a Safety-Critical System Based on Coordinated Atomic Actions. In *Proceedings of the 29th Int. Symp. on Fault-Tolerant Computing* (FTCS-29), IEEE CS Press, Madison, WI, USA, June 1999.

[24] A. Zorzo, A. Romanovsky, J. Xu, B.Randell, R.J. Stroud, I. Welch, *Using Coordinated Atomic Actions to Design Complex Safety-Critical Systems: A Production Cell Case Study*. In Software: Practice and Experience, 29(8), pp. 677-697, 1999.

# Supervisory Plug-ins for Distributed Software

Michael Lemmon and Kevin X. He

Dept. of Electrical Engineering
University of Notre Dame, Notre Dame, IN 46556
`lemmon,xhe@maddog.ee.nd.edu`

**Abstract.** This paper demonstrates the use of supervisory control theory in synthesizing plug-ins for distributed software. The plug-ins are software objects that *supervise* an existing distributed system so that certain properties such as fairness and deadlock freedom are guaranteed. The distributed application is modeled as bounded ordinary Petri net and system analysis is accomplished through a partial order method known as unfolding. The unfolding constructs an event structure that provides a natural encapuslation of concurrent threads of execution whose selective disablement by the supervisory plug-in assures the desired application property. The synthesis of the plug-in is based on results from supervisory control theory and the synthesized plug-ins are ”optimal” in that they are maximally permissive. We demonstrate our approach on a distributed cache system.

## 1 Introduction

Distributed software may be viewed as a collection of objects that interact through message passing. Such software is of growing importance and it appears in applications such as military command and control, commercial telecommunications, and emergent Internet applications. Distributed software is required in networks of embedded processors that are used to control major components of the national infrastructure such as the electric power grid and air traffic control system. Due to the critical nature of such applications, it is essential that we develop systematic methods for assuring the quality of such distributed software.

Assuring the quality of distributed software can be extremely difficult. Difficulties arise due to the concurrent and decentralized nature of the applications. The ”open” nature of many distributed software architectures also introduces many challenges. As an example, consider a network controlling an electric power distribution system. This system consists of older (so-called *legacy*) hardware and software components, as well as newer components. The open nature of the architecture allows newer components to enter and leave the system freely. For such systems we need to ensure that the entry or departure of newer components does not interfere with the performance of the legacy components. Assuring re-usability of legacy components under software upgrades can be difficult for a variety of reasons. In the first place, the concurrent nature of the applications means that analyses of the overall system are hampered by state-space explosion. In the second place, these systems can be dynamic in that objects may enter or leave the system at run-time. In such a dynamic environment it can be extremely difficult to analyze system behavior due to uncertainty in the suite of objects comprising the current system. In the third place, the mandate for open software architectures means that designers have limited control over legacy components, thereby complicating the design process. Finally, newer and older objects may interact in unpredicatable ways that introduce new or emergent patterns of behavior. Such emergent behaviors can be very difficult to identify due to the large scale nature of the system's state space.

Solving the preceding problems requires a formal and scalable analysis method for distributed systems. This paper presents a formal approach to the design of distributed software that applies methods from supervisory control theory [1]. Supervisory control theory is concerned with the regulation of discrete-event systems. It provides a clear characterization of optimal supervision. In this paper, we use this theory to synthesize *plug-ins* to a distributed application that enforce specified properties such as fairness or deadlock freedom. The underlying formal model is a bounded Petri net. We analyze the net's behavior by a partial order method known as unfolding [2]. Unfolding transforms the original system into an event structure from which we can encapsulate concurrent threads of execution that can be selectively disabled by the synthesized plug-ins to enforce the desired behavioral properties [13]. The use of supervisory control theory ensures that the resulting system is *optimal* in the sense of being maximally permissive. We demonstrate our ideas on a distributed cache system.

**Remark:** While the results in this paper only pertain to bounded Petri nets, it should be noted that many of the definitions also apply to unbounded networks as well. In fact, there is every reason to believe that the underlying principles advocated in this paper can also be applied to restricted classes of unbounded nets. What has not been demonstrated for unbounded nets, however, is the optimality or existence of such supervisors.

The remainder of this paper is organized as follows. Section 2 articulates the supervisory control problem for distributed software. Section 3 discusses the synthesis of supervisory plug-ins using a partial order method known as unfolding. Section 4 shows how these ideas can be applied to the the run-time reconfiguration of distributed software. Section 5 summarizes the principal results and future directions of this work.

## 2 Supervising Distributed Software

This section articulates a framework in which to pose the problem of designing *supervisory plug-ins* for a distributed application. By a *supervisory plug-in*, we mean a software object that can be composed with the base application to restrict the base application's behavior without adding any new behaviors. This restriction of the original system behavior is referred to as the *supervisory control problem* in the control systems literature. We therefore pose the problem of designing supervisory plug-ins as a supervisory control problem.

The underlying design paradigm in this paper is illustrated in figure 1. For the moment, ignore the details in this figure and focus on the feedback connection. We see that this system consists of the base application software (what we refer to as the *plant*) and the plug-in component (what we refer to as the *supervisor*). The supervisor uses information from the plant to control the base application's behavior. As noted in the first paragraph, this control actually restricts or disables the plant from executing certain actions that are considered undesirable. The ”restrictive” nature of this control is what control theorists refer to as a ”supervision” (as opposed to regulation).

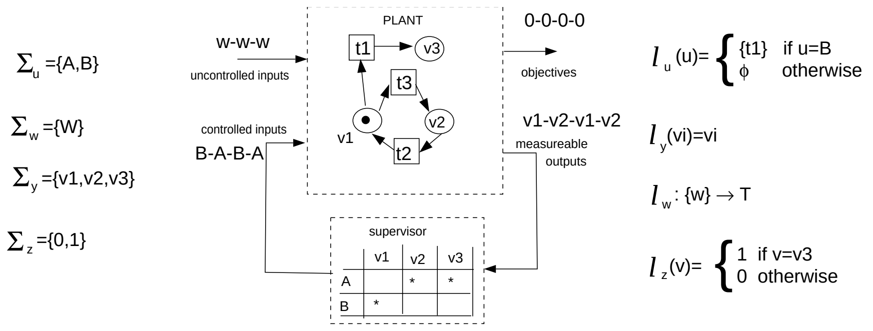


**Fig. 1.** Supervisory Control Loop

From a pragmatic point of view, there are many ways in which the block diagram in figure 1 can be viewed. One view treats the supervisor as a mobile software object that the user downloads to a pre-existing object in the base application. This view is particularly useful in dynamically reconfigurable software [3] [4] where plug-ins are composed with the base application at run-time in order to take advantage of changes in the structure of the application. As noted earlier, such changes can occur frequently in network based applications. In another, more conventional viewpoint, the ”plug-ins” become patches to the original program. These patches are synthesized to correct ”bugs” in the original application software. In this case, the block diagram in figure 1 is an archetype for the software design process, in which we iteratively analyze and correct ”bugs” in a given application.

The loop in figure 1 is the classical control loop used in all of traditional control theory [5]. We now begin filling in the details of figure 1. In the loop, the plant $P$ is treated as a *two-port* system. In other words, the plant is a system with two types of inputs and two types of outputs. Plant inputs are categorized as either being *control signals* or *uncontrolled disturbance signals*. The control inputs are chosen by the designer/supervisor to regulate plant behavior. The uncontrolled disturbance input is not completely known by the designer. The disturbance is really used to enable a set of possible next events that the plant can generate. In this regard, therefore, the uncontrolled disturbance injects some degree of non-determinism into the plant state’s evolution. The system outputs are also categorized into two distinct groups. The *objective* signal is an output that quantifies the system’s performance. In our case, we take the object signal to be a ”warning” that the system has entered a forbidden state. The other output signal is called the *measurement* signal. This signal is directly observed by the supervisor and is used by the supervisor to help direct the plant’s behavior.

With the loop in figure 1, we now associate the *supervisory control problem*. In particular, our problem is to find a *supervisor* that prevents the plant from entering any previously specified forbidden state. In general, there may be a number of supervisors that accomplish this objective. For instance, if we have a supervisor that disables all plant transitions, then we would have achieved the objective (assuming we started in a safe state to begin with). Such solutions to the supervisory control problem, however, are highly undesirable because they are extremely restrictive. In particular, we would like to determine a *maximally permissive* supervisor. If such a maximally permissive policy exists, then we say it is ”optimal”. Supervisory control in traditional control of discrete event systems is concerned with the existence of and method for synthesizing the maximally permissive (i.e., optimal) supervisor. The application of this theory to software design means that we can clarify what it means for a particular software solution to be *optimal*.

The preceding discussion is still somewhat informal. We now provide a complete formal description of the supervisory control problem. We begin by considering the base application. Let’s assume that the plant can be represented by a net system $G = (N, \mu_0)$ where $N = (S, T, F)$ is an ordinary bounded Petri net with places $S$, transitions $T$, and directed arcs $F \subset (S \times T) \cup (T \times S)$. $\mu_0$ is the initial marking of the net system.

For notational purposes, let’s review some basic Petri net concepts. The current state of the Petri net is represented by the *marking* $\mu : S \to Z^+$. $\mu$ maps each place in the Petri net onto a non-negative integer. If $\mu(s) > 0$, then we say that place $s$ is marked. Given a transition $t \in T$, we define the preset of $t$ (denoted as $\bullet t$) as all places $s \in S$ such that $(s,t) \in F$. Similarly, the postset of transition $t$ (denoted as $t\bullet$) is the set of all places $s \in S$ such that $(t,s) \in F$. A marking $\mu$ is said to be reachable from $\mu_0$ through transition $t$ (also denoted as $\mu_0 \xrightarrow{t} \mu$) if and only if $\mu(s) > 0$ for all $s \in \bullet t$ and

$$\mu(s) = \begin{cases} \mu_0(s) + 1 \text{ if } s \in t\bullet - \bullet t \\ \mu_0(s) - 1 \text{ if } s \in \bullet t - t\bullet \\ \quad \mu_0(s) \qquad \text{otherwise} \end{cases}$$

A string of transitions $\sigma = t_1, t_2, \cdots, t_n$ is called an *occurrence sequence* if there exists a sequence of marking vectors $\mu_0, \mu_1, \cdots, \mu_n$ such that $\mu_{i-1} \xrightarrow{t_i} \mu_i$ for $i = 1, \ldots, n$. The set of all markings reachable from $\mu_0$ is denoted as $R(\mu_0)$.

To embed the net system $G = (N, \mu_0)$ into the control loop in figure 1, we define an *augmented plant* as the 5-tuple,

$$P = (G, \ell_w, \ell_u, \ell_z, \ell_y)$$

where $\ell_w : \Sigma_w \to \text{Pow}(T)$ maps symbols in the disturbance alphabet $\Sigma_w$ onto a set of transitions, $\ell_u : \Sigma_u \to \text{Pow}(T)$ maps symbols in a control alphabet $\Sigma_u$ onto a set of transitions. $\ell_u$ and $\ell_w$ are input functions. A transition $t$ is said to be *uncontrollable* if and only if there exists no control symbol $\lambda$ such that $t \in \ell_u(\lambda)$. The output maps $\ell_z : R(\mu_0) \to \Sigma_z$ and $\ell_y : R(\mu_0) \to \Sigma_y$ map the net system’s current marking onto a symbol in an objective alphabet, $\Sigma_z$, or measurement alphabet, $\Sigma_y$, respectively.

The *supervisor* is a map $S : \Sigma_y \to \Sigma_u$ from the measurement symbols to the control symbols. The supervised plant $P|S$ is the interconnection of the augmented plant and supervisor shown in figure 1. We can view this supervised system as an input/output system that accepts a string of disturbance symbols and generates a string of objective symbols. In particular, consider a sequence of objective symbols $\sigma_w = w_1, w_2, \cdots, w_n$.

We say that the occurrence sequence $\sigma = t_1, t_2, \cdots, t_n$ is *accepted* by the supervised plant $P|S$ with input sequence $\sigma_w$ if and only if there exists a sequence of reachable markings $\mu_0, \mu_1, \cdots, \mu_n$ such that

– $\mu_0 \xrightarrow{t_1} \mu_0 \xrightarrow{t_2} \cdots \xrightarrow{t_n} \mu_n$

– $t_i \in \ell_w(w_i)$ for all $i = 1, \ldots, n$, and
– $t_i \notin \ell_u(S(\ell_y(\mu_{i-1})))$ for $i = 1, \ldots, n$.

The sequence of objective symbols $\sigma_z = z_0, z_1, z_2 \cdots, z_n$ is generated by this occurrence sequence if $z_i = \ell_z(\mu_i)$. From these definitions, we see that the action of the supervisor is to *disable* specified transitions from firing when the net system reaches a given marking in the domain of $\ell_u$. Because supervision is based on the marking of the plant's Petri net, we refer to $S$ as a *marking based supervisor*.

The system in figure 1 represents a specific supervisory control system in which plant's network, $(S, T, F)$ has the form shown in the figure. The output alphabets are $\Sigma_y = \{v_1, v_2, v_3\}$ and $\Sigma_z = \{0, 1\}$. In this example, we define the input maps so that $\ell_u(B) = \{t_1\}$ and $\ell_u(A) = \emptyset$. The disturbance input map is $\ell_w(w) = T$. With these definitions, we see that the control input only disables transition $t_1$ when the control symbol is $B$. Moreover, we see that the disturbance input was chosen so that at any instant, all transitions that have their presets marked (and which have not been disabled by the supervisor) will be free to fire. The output map $\ell_y$ is chosen so that the supervisor can see all markings generated by the system. The other output mapping, $\ell_z$, generates a 1 if the marked state is $v_3$. Note that $v_3$ is a deadlocked place from which the net system cannot fire another transition. Therefore the objective map warns us when the system is deadlocked. The supervisor shown in figure 1 is represented as a table for the supervisor map. It shows that we generate the output symbol $B$ when the system marks state $v_1$. Since $v_1$ has a transition, $t_1$, leading to the deadlocked marking $v_3$, it is apparent that the action of the supervisor is to prevent the system from reaching the deadlocked state. This is accomplished by disabling $t_1$ from firing if place $v_1$ is marked. We therefore see that the supervisor, $S$, in this example is a deadlock-avoidance supervisor.

The preceding discussion shows a supervisor that prevents the plant from reaching deadlocked marking. The deadlock avoidance property is a *specification* on the closed loop system's desired behavior. We now generalize the ideas presented in the preceding paragraph. Let the subset $R_f \subset R(\mu_0)$ be a collection of *forbidden markings* in the original net system. Assume that $\ell_z$ is chosen to signal the system's entry into a forbidden marking. In other words, let $\ell_z(\mu) = 1$ if and only if $\mu \in R_f$. Let's partition the transitions $T$ into a set of *controllable* $T_c$ and *uncontrollable* transitions. Recall that an uncontrollable transition is a transition that cannot be disabled by the supervisor. Therefore if $t \in T_u$, we know that $t \notin \ell_u(u)$ for any $u \in \Sigma_u$. Let $L(G)$ denote the set of all occurrence sequences that can be accepted by $G$. Let $K$ be a sublanguage of $L(G)$ (also called the specification language) such that all occurrence sequences in $K$ generate objective sequences that have no ones $(1)$ in them (i.e., no forbidden markings are entered). Consider the supervised plant $P|S$ and let $L(P|S)$ denote the set of all occurrence sequences in $L(G)$accepted by $P|S$ for any input sequence $\sigma_w \in \Sigma_w^*$. The supervisor $S$ is said to be *legal* if $L(P|S) \subseteq K$. We say that the supervisor is *maximally permissive* if for any other legal supervisor $S'$, then $L(P|S') \subseteq L(P|S)$. The *supervisory control problem* asks us to find the maximally permissive legal supervisor. The language generated by this maximally permissive supervisor is denoted as $K^{\uparrow}$ and is called the *supremal controllable sublanguage*.

Since a bounded ordinary Petri net can be represented as a finite state machine, the results of [10] can be used to infer the existence of the supremal controllable sublanguage. In other words, the supervisory control problem always has an *optimal* solution (optimality being interpreted in the sense of maximal permissivity) and this means that our problem statement is well-posed. The chief contribution of supervisory control theory is to ensure that the problem of finding an optimal supervisor for a discrete-event system is meaningful.

Let's consider a specific example that illustrates the application of this framework to the design of supervisory plug-ins for distributed software. We consider a distributed cache system in which two local cache memories synchronize their contents to the contents in a global memory bus. Each local cache has memory and a processor to control memory access. The global bus also has memory and a processor. The global bus and local caches interact through message passing.

We assume that the local cache and global bus each have three states: *invalid*, *shared*, or *owned*. When the data in the local cache and global bus are synchronized then all objects are in the *shared* state. The data in a cache becomes unsynchronized when the local cache updates its memory. When a local cache wants to perform this update, it changes its local state to `owned` and sends the message `invalidate` to the other cache and global bus. Upon receipt of this message the global bus and other cache change their internal states to `invalid`.

The local cache can also invalidate the global bus if it detects a loss of synchrony through a `read-only` message. This provides a means of fault identification for memory faults. In this case, the local cache issues a `read-only` request to the global bus. The global bus responds with the requested data and if this data is inconsistent with the local cache's copy, then the local cache issues an `invalidate` signal.

Once invalidated, the cache memories need to be resynchronized to the global bus. This resynchronization is initiated by the local cache that changed its local memory. After updating its memory, the local cache sends the `complete-update` message to the global bus. The global bus responds with a `read-write` request for the updated data. The `owned` local cache responds to this request with the desired data and then changes its internal state to `invalid`. Upon receipt of the data, the global bus changes its state to `owned`. We now have both local caches invalidated and they can each change their internal state to `shared` by issuing `read-write` requests to the global bus.

The software controlling the communication between the local caches and the global bus can be viewed as distinct objects. We will formally model these objects as ordinary Petri nets. For instance, let $N_{lc} = (S, T, F)$ be a Petri net for one of the local caches. The messages sent and received by this object are a subset $S_{\text{meas}}$ of $S$ such that if $s \in S_{\text{mes}}$, then either $s\bullet = \emptyset$ or $\bullet s = \emptyset$. In other words, messages are places that are either sinks or sources of the object Petri net. We define the *private variables*, $S_{\text{priv}}$, of the object as all other places (i.e. $S_{\text{priv}} = S - S_{\text{mes}}$). The *public methods* of the object are transitions that are connected to message places. In particular, we define the *i*th public method as a subset $M_i \subset T$ such that if $t \in M_i$ then

– $\bullet t - S_{\text{mes}} \neq \emptyset$ or $t \bullet - S_{\text{mes}} \neq \emptyset$
– and if for any $t_1, t_2 \in M_i$, we know $\bullet t_1 - S_{\text{mes}} = \bullet t_2 - S_{\text{mes}}$ and $t_1 \bullet - S_{\text{mes}} = t_2 \bullet - S_{\text{mes}}$.

In other words, public methods are collections of transitions whose arcs to message places all have the same connectivity pattern. Petri net objects for the local cache and global bus objects are shown in figure 2. In this figure, private places are represented by open circles. Note that in figure 2, arcs going to (from) message places do not terminate (originate) in an open circle. Message places are not explicitly shown in this figure. Also note in this figure that all transitions are public method. This, however, is not always the case as we can define objects that have internal private transitions.

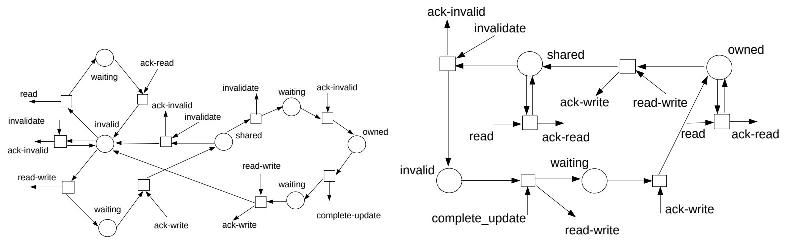


**Fig. 2.** Distributed Cache System Object Petri Nets (local cache on left, global bus on right)

We now formulate the supervisory control problem for the distributed software system comprised of the objects in figure 2. Recall that the formulation of the problem starts with a specification of the system's forbidden markings. One obvious set of forbidden markings would consist of those markings from which the entire system is *deadlocked*. Assuming that we can identify the deadlocked markings $M_{\text{dead}}$, (this is done below using a partial order method), we can then define the objective map $\ell_z$ to take values of 1 when $\mu \in M_{\text{dead}}$ and zero otherwise. We're interested in designing another object (called a *plug-in*) that can selectively disable system transitions to enforce deadlock-avoidance.

Since the plug-in is an object as well, it can only interact with the other objects through their public methods. Therefore, our supervisor can only disable those transitions that are found in public methods. In this particular example, all of the transitions are in a public method and so we have complete controllability over this system. However, this need not be the case in general. For objects in which there are transitions representing private methods, these transitions cannot be directly disabled. The *uncontrollability* of various transitions in the network makes the problem of identifying a supervisory controller much more difficult. One of the chief accomplishments of supervisory control theory was the articulation of a framework for such uncontrollable net systems and the identification of necessary and sufficient conditions for the existence of legal controllers enforcing a desired specification language.

It is important to note that the *supervised* net system is not an ordinary Petri net. The supervisor disables transitions when a specified set of places are marked in the plant. This disabling action cannot always be represented by a Petri net. It is well known [6] that Petri net languages are not closed under the recursive operation used to compute the supremal controllable sublanguage. This result, therefore, implies that an optimal (maximally-permissive) marking based supervisor does not necessarily have a representation as an ordinary Petri net. On the basis of this result, therefore, we can partition results on supervisory control into marking-based supervisors and Petri-net based supervisors [7] [8] . The framework presented above uses a marking-based supervisor and in this case, we know a maximally permissive supervisor can always be found. There are obvious benefits in being able to represent the supervisor as a Petri net. For some classes of specifications (deadlock), necessary and sufficient conditions for the existence of a marking-based supervisor and Petri-net based supervisory have recently been proposed [9].

## 3 Synthesis of Supervisory Plug-ins

Most existing approaches [11] [12] [13] for the synthesis of maximally permissive supervisors involve some sort of search of the Petri net's state space. The problem with this, of course, is that distributed software has a high degree of concurrency, so it's impractical to search exhaustively for critical transitions leading to forbidden markings. This means that clever and efficient means of search must be employed if we are to automate the design of supervisory plug-ins for this class of software. This section summarizes recent results in [13] in which a partial order method known as network unfolding is used to synthesize marking based supervisors. Partial order methods [14] [15] [16] represent an important approach that can greatly reduce the complexity of network analysis. This point was first made in [17] where unfolding was proposed as a means taming state-explosion problems encountered in the verification of asynchronous digital circuits. Since that time a variety of researchers have used unfolding [20] [18] [19] to characterize network properties and in [13], this idea was extended to synthesize supervisors enforcing this set of characterized properties. In this section we summarize the approach to supervisor synthesis and then illustrate its application to the design of supervisory plug-ins that enforce deadlock-freedom for the distributed cache example in the preceding section.

Consider a net system $(N', \mu_0')$, where $N' = (S', T', F')$ is an ordinary Petri net and $\mu_0'$ is the initial marking. Let $\min(N')$ be those places in $N$ with empty presets. We define an *occurrence net* as a net system $(N', \mu_0')$ such that every place is preceded by at most one transition (i.e., $|\bullet s| \leq 1$ for all $s \in S'$), no transition is in self-conflict and a place $s \in \min(N')$ if and only if it is marked by $\mu_0'$. A *branching process* $\beta = (N', h')$ of net system $N$ consists of an occurrence net $N'$ and a net homomorphism, $h'$, from $N'$ to $N$. The net homomorphism preserves the causal ordering of transitions. Specifically this means that if $\bullet t_1 = \bullet t_2$ and $h'(t_1) = h'(t_2)$ then $t_1 = t_2$. In general a given network may have many branching processes and the unfolding is the maximal branching process (denoted as $\beta_m$).

The unfolding of a deadlock-free net system will always be of infinite size. Nonetheless, it is often possible to find a branching process $\beta_c$ that is a finitary prefix of the unfolding $\beta_m$ such that all the reachable marking of the original net system $N$ can be enumerated from $\beta_c$. An important prefix of this type was introduced in [17] and subsequent variations were presented in [20]. A key concept in the characterization of such prefixes is the concept of a *configuration*. Let $N_m$ be the occurrence net associated with the unfolding $\beta_m$. A set of transitions, $C$, is said to be a configuration if and only if all transitions in $C$ are in precedence and no two transitions in $C$ are in conflict.. Given a transition $t \in T$, we define the *cause* of $t$ (denoted as $[t]$) as the set of all transitions preceding $t$. The cause is easily shown to be a configuration.

The cut of a configuration $C$ is defined as

$$\mathrm{Cut}(C) = (\min N' \cup C\bullet) - \bullet C$$

where $C\bullet$ ($\bullet C$) is the postset (preset) of transitions in $C$. The cut of the configuration contains those places that are marked after firing all transitions in $C$.

Given an unfolding $\beta_m$, we define the set of *end transitions* $T^{\mathrm{end}}$ as any transition such that

– there is no transition $T' \in T_m$ such that $t < t'$,
– or there exists a transition $t' \in T$ such that $[t'] \subset [t]$ and $h_m(\mathrm{Cut}([t'])) = h_m(\mathrm{Cut}([t]))$.
– or the marking of $h_m(\mathrm{Cut}([t]))$ is the initial marking $\mu_0$.

The transitions in $T^{\mathrm{end}}$ represent a natural place to cut the unfolding because these transitions either represent local deadlocks (the first condition) or they represent transitions that enable other configurations within the net system (the final two conditions). We define the branching process $\beta_c$ by removing those nodes in the unfolding that follow transitions in $T^{\mathrm{end}}$. It has been shown that $\beta_c$ is a finitary prefix that enumerates all reachable markings in $R(\mu_0)$. In this regard, we can construe the net system $N_c$ associated with $\beta_c$ as a reduced reachability graph.

Recognizing the importance of $T^{\mathrm{end}}$, we refer to the cause of any transition $t \in T^{\mathrm{end}}$ as a *base configuration*. Base configurations encapsulate causally related strings of transitions and as such they represent a thread of execution that can be seen, intuitively, as basis threads from which all other system behaviors can be generated. Another way of viewing the base configuration is as a *meta-state* for the system. From the standpoint of supervisory control, these meta-states can be selectively disabled to enforce, in a modular manner, various specifications on the system behavior. The unfolding process provides an a means of automatically identifying these meta-states as we construct the net system's reduced reachability graph. The unfolding process also allows us to easily identify the markings enabling these meta-states. This means that unfolding can be readily used to synthesize supervisory controllers for Petri nets, a fact that was first advocated in [13].

As an example, let's consider the Petri net shown in figure 3 and construct its unfolding. The original network, $N$, is shown in the top part of this figure with an initial marking of places $s_1$ and $s_3$. The associated occurrence net for $N$ is shown in the second graph in figure 3. We construct $N'$ from $N$ by following the markings that are reachable from the initial marking. In figure 3, the first transition that is enabled is transition $t_3$ and this results in the marking $\{s_2, s_4\}$. We construct the network associated with this transition as shown in the *first tier* of the occurrence net in figure 3. The second tier is constructed by considering the markings reachable from $\{s_2, s_4\}$. In this case, there are two possible transitions that can be enabled, $t_1$ and $t_4$. However, the network has a choice in which transition is fired. The mappings reached by choosing either of these transitions forms the *second tier* of this particular unfolding. Note that our unfolding has now identified two different paths that the Petri net can follow. These represent two different concurrent executions of the system. The final *third tier* of the unfolding is obtained by firing the transitions $t_2$ and $t_4$.

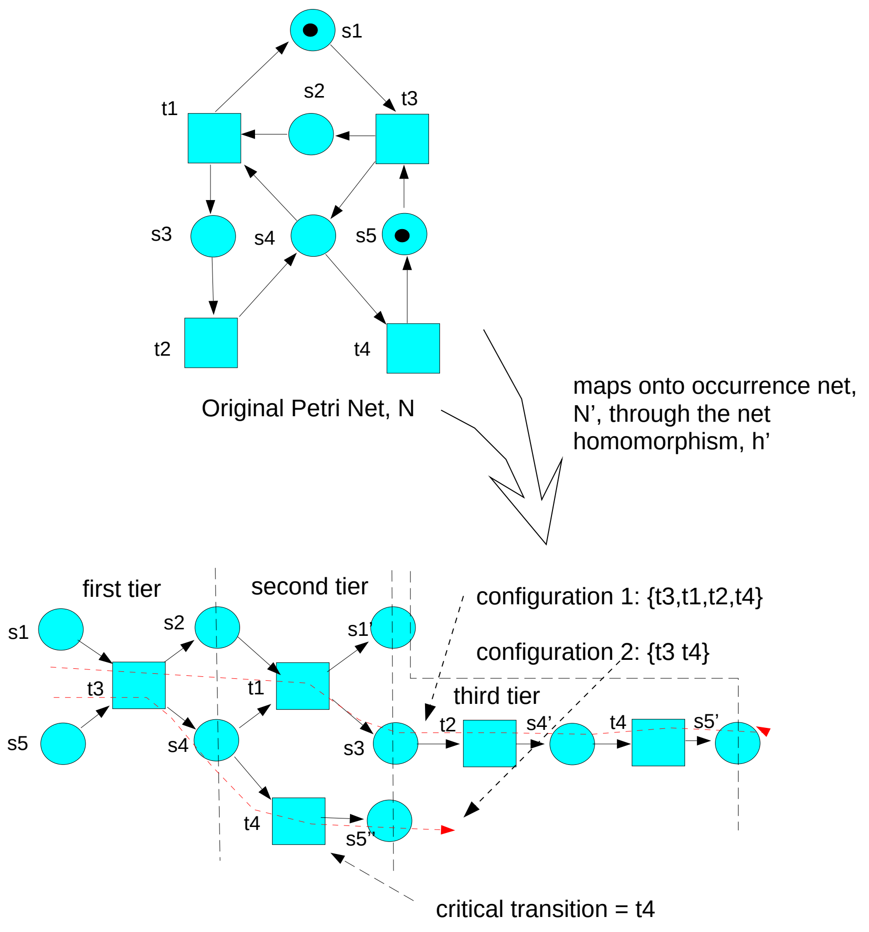


**Fig. 3.** Unfolding of Petri Net

Since unfolding preserves precedence relations between transitions, it can be used to identify *critical transitions* whose controllability ensures the existence of a maximally permissive supervisor disabling a specified base configuration. In figure 3, we have the finite prefix $\beta_c$ for the unfolded system. In this unfolding there are two base executions. These base configurations are formed from the set of transitions $BC_1 = \{t_1, t_2, t_3, t_4\}$ and $BC_2 = \{t_2, t_4\}$. The first execution $BC_1$ is a cycle, in that upon completion of the execution, the network reaches a marking from which $BC_1$ can be re-enabled. The second execution, $BC_2$, is deadlocked. The unfolding in figure 3 explicitly shows how the network can be deadlocked when it chooses to execute base configuration $BC_2$. In order to prevent deadlock, we simply need to find that transition which disables $BC_2$ from being executed while keeping configuration $BC_1$ alive.

In figure 3, it is apparent that the critical transition disabling $BC_2$ is transition $t_4$. This transition can be fired in $BC_2$ when the net marking is $\{s_2, s_4\}$. If we then introduce a supervisory map, $S$, that disables $t_4$ when these two places are marked then we can ensure that the deadlocked base configuration cannot be executed. Moreover, by disabling the execution of $BC_2$ and noting that $BC_1$ contains all transitions of the original net system $N$, we see that this proposed supervisor in fact enforces the liveness of the supervised system. It is, of course, crucial, in this example that $t_4$ be controllable. If $t_4$ is not controllable, then it may be possible to look at those controllable transitions preceding the critical transition and see if disabling any of them will achieve the same result (namely disabling $BC_2$ and keeping $BC_1$ alive). In this example, unfortunately, there are no such controllable transitions preceding $t_4$ and therefore the controllability of transition $t_4$ is necessary and sufficient for the existence of a supervisory policy enforcing system liveness.

We now apply these ideas to the supervisory control of the distributed cache system. This particular system has a deadlock that is relatively difficult to detect. We want to synthesize a plug-in that makes the system deadlock free. Using the unfolding methods in [19], we find that the distributed cache system is deadlocked whenever one local processor sharing data with the global bus is trying to invalidate the other local cache's memory, and at the same time the other cache is requesting to read data from the global bus. The configuration in figure 4 shows how the deadlock occurs. Local cache 1 sends out the *invalidate* message and awaits acknowledgement from local cache 2. In the meantime, local cache 2 sends out a `read` request to the global bus and awaits its response. The global bus, however, cannot respond because its memory was invalidated by local cache 1. In this case both local caches and the global bus cannot proceed and the system is deadlocked.

The situation illustrated in figure 4 is a race condition that is referred to as *cyclic lock* in [19]. Cyclic locks occur when a sequence of transitions in concurrent base configurations are interleaved in such a way that both configurations are waiting for resources that the other configuration needs to release. The sequence of transitions leading up to this race condition is called a *lock sequence*. Lock sequences can be identified during the algorithmic construction of the net system's unfolding. The data in the unfolding can also be used to identify those markings that must be disabled in order to prevent the lock sequence from firing. It therefore becomes possible to use the unfolding method

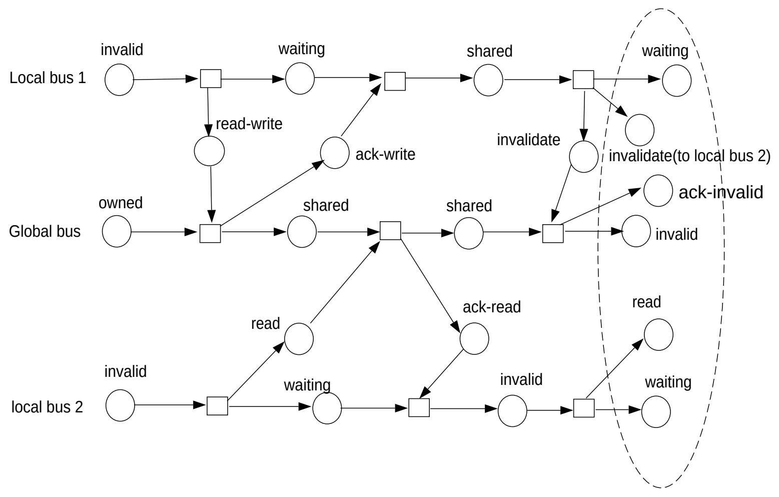


**Fig. 4.** Race Condition in Distributed Cache System

to synthesize a supervisory controller that enforces deadlock freedom in a maximally permissive manner.

In the distributed cache example, our plug-in must not let the global bus invalidate its memory if there is a read request in its message queue. By disabling this transition, we force the global bus to respond to the `read` request sent by the other local cache. On the other hand, we must also disable a local bus' read request if there is an `invalidate` request on the local cache's message queue. This restriction forces the local cache to acknowledge the invalidation request and let the data update proceed.

The implementation of this supervisory action is rather simple. We only need to develop two types of supervisory plug-ins, one for the global bus and the other for the two local caches. Pseudo code for the global bus plug-in is

```
if( (Private_State==SHARED) &&
    (message_queue(invalidate)) &&
    (message_queue(read)) )
disable(Shared2Invalidate)
```

In this pseudocode, the variable `Private_State` is the global bus state and the function `message_queue` checks the message queue for the specified message. If the conjunction of these conditions is true, then the function `disable` disables the invalidation of the global bus' memory. Pseudo code for the plug-ins on the local caches is

```
if( message_queue(invalidate)){
    Disable(Send_read_request);
} else {
    Enable_all();
}
```

This pseudocode simply tests to see if the local cache has an `invalidate` message on its message queue. In which case, the `read_request` is disabled. If the message queue does not have an `invalidate` message queued up, then the `read_request` action is re-enabled.

## 4 Runtime Reconfiguration of Distributed Software

This section speculates on the application of supervisory control to the runtime reconfiguration of distributed software. Due to the open and dynamic nature of the physical layer in embedded network systems, there is a real need for distributed software that can monitor its own health and then autonomously reconfigure itself to improve its performance. This is the notion of *dynamic reconfigurable* or *adaptive* software [3] [4].

The underlying paradigm is shown below in figure 5. Ignoring the details in this figure, we see that the basic control loop of figure 1 has been embedded into a larger control loop. The objective signal is now used by a *switching element* to select which supervisors are to be applied to our software system. To use the switching element we must monitor the behavior of the augmented plant to detect anomalous behavior. Upon detection of an anomaly that adversely effects system performance (as represented by the objective symbols), the switching element reconfigures the software system by selecting a different set of supervisory plug-ins to control the plant. This reconfiguration, of course, is done at run-time using pre-compiled objects that are simply ”plugged-into” the augmented plant.

A more detailed examination of figure 5 shows how the monitoring is actually accomplished. We see that the objective signal is passed through a mapping $Q : \Sigma_z^* \to \Re$. $Q$ maps each string of objective symbols onto a real number that represents the *Quality of Service* or *QoS* provided by the plant for this particular input sequence. The QoS is a real number and can represent a number of practical performance measures. Returning to our distributed cache example, one useful measure of QoS is the time it takes for the system to return to the `shared` state after the global bus has been invalidated. If this resynchronization time is too long, then this means something is wrong within the system and our switching element is used to reconfigure the software. The reconfiguration decision is made by a simple threshold test on the length of the resynchronization time as output by $Q$. So in figure 5, we see that the map $Q$ is followed by a thresholding element that provides a binary output to the switching element indicating whether or not the system needs to be reconfigured.

The actual nature of the reconfiguration depends on the suite of supervisory plug-ins we have at our disposal. In the preceding section, it was shown that this system has a cyclic lock that can be fixed through a deadlock-avoidance plug-in. It is also possible to introduce plug-ins for other specifications. In this distributed cache example, we consider a *fairness* specification.

To motivate this fairness specification, let’s assume that one of the local caches issues an `invaliate` message, but for some reason the cache is unable to complete updating its memory. This may happen due to a processor fault. If this happens, then `read` requests from the other local bus will be blocked because the global bus has been invalidated. However, since the local cache never issues the `complete-update`

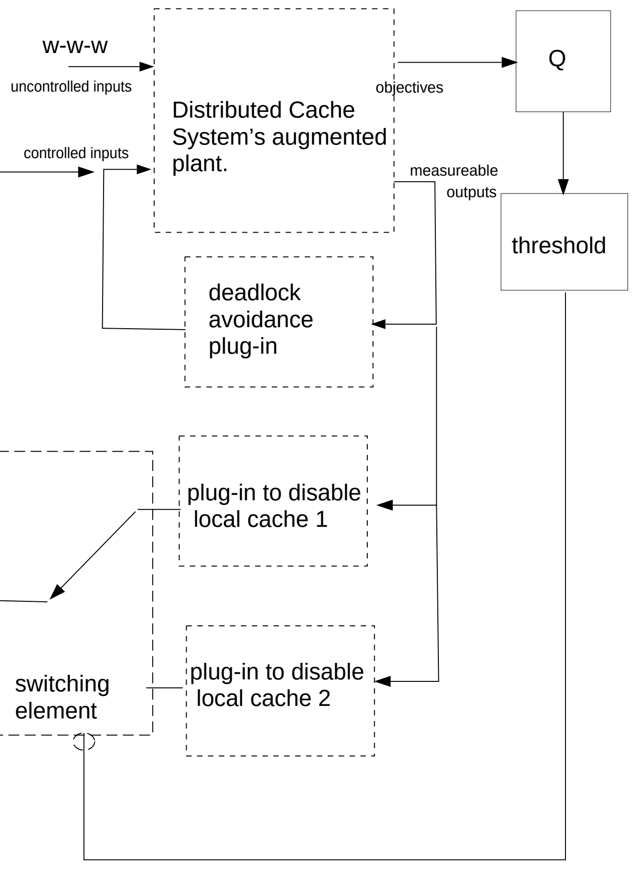

w-w-w
uncontrolled inputs
controlled inputs
Distributed Cache System's augmented plant.
objectives
Q
measureable outputs
threshold
deadlock avoidance plug-in
plug-in to disable local cache 1
plug-in to disable local cache 2
switching element

**Fig. 5.** Runtime Reconfigurable Distributed Software

message, the global bus is blocked from leaving the `invalid` state. This type failure in the local cache, therefore, is sufficient to deadlock the entire distributed cache system. In other words, our distributed software system is not fault tolerant since it fails globally when a single local cache is faulty.

We want to fix this problem without rewriting the existing protocols in the local caches and global bus. We solve our problem by introducing a plug-in that can be applied when this type of fault is detected. Obviously, this fault results in extremely long (i.e. unbounded) resynchronization times, so the simple threshold test mentioned above can be used to detect this type of fault. The most obvious action to be taken at this time is to simply isolate the faulty local cache from the entire system. This is accomplished by introducing a supervisory plug-in on the faulty cache that disables the transitions sending *invalidate* messages. In figure 5, the additional two supervisors connected to the switching element are the plug-ins disabling these transitions.

**Remark:** Note that in addition to applying plug-ins to isolate the faulty cache, we must also reinitialize the global bus as well. Recall that the deadlock induced by this fault leaves the global bus in a state from which it is deadlocked unless a `complete-update` message is received. By simply disabling messages from the faulty cache, however, we do not clear this deadlock. It is therefore essential that in addition to turning on the plug-ins, that the global bus is re-initialized. This re-initialization is not shown in figure 5. However, it must be realized that re-initialization is an important part of dynamic software reconfiguration.

**Remark:** One important aspect of supervisory control in this application is the apparent composability of the supervisors. Our early work indicates that these supervisory plug-ins can be composed in such a manner that they do not interfere with each other. This is obviously apparent in the distributed cache example, where we can apply the deadlock-avoidance and fairness plug-ins without losing either property. Whether or not this composability is a general property of supervisory plug-ins is currently being investigated and will be reported upon in the future.

## 5 Conclusions

The primary contribution of this paper is the proposed application of supervisory control theory to the synthesis of supervisory plug-ins for distributed software. This theory ensures that the synthesis problem is a well-posed optimization problem in which we search for maximally permissive marking-based supervisors. The theory applies to bounded net systems with uncontrollable transitions and this means that it is relevant to open architecture software systems where a designer has limited access to object methods. Moreover, recent advances in partial order method analyses provide systematic methods for the synthesis of such supervisors for certain classes of specifications such as deadlock and fairness. In short, the methods presented in this paper apparently provide a systematic and tractable set of methods that may be used to automate the design of high quality distributed software. This conjecture was exemplified by using the framework to formulate an approach to run-time reconfigurable software.

This paper is a *concept* paper and there remain a number of important issues that must be addressed before this concept can be implemented in practice. Some of these issues are itemized below.

– This paper's restriction to bounded Petri net immediately suggests that traditional finite-state machine methods for verification and supervision might be applied as well. The potential benefit that partial order methods bring to this analysis is a reduction in the analysis' complexity. Even though this paper has not formally quantified that reduction in complexity, it is possible to speculate that the computational savings obtained using this method will vary greatly with the complexity of the process being studied. Systems having a few sparsely connected fundamental cycles (such as the dining philosopher's problem) are well served by this method. Other problems having many densely connected fundamental cycles may be better served

using the binary decision diagrams employed in the verification of finite-state machines. Future work is needed to quantify where and when partial order methods are most valuable.
- There is an important question concerning the implementation of the supervisor in a distributed system. Clearly, the supervisor requires access to at least a partial global state before it can disable a method. Identifying such states in a distributed system can be extremely difficult. One approach that has been suggested is for supervisors to use time-stamped messages to construct a partial system state that is known to be valid at a specified time in the past. The firing of object methods, therefore, must also accomodate such a delay. This is, probably, a function to be implemented in network middleware. These ideas are also being explored by our group.
- Our recent work in this area suggests that supervisory approaches indeed provide a method for composing software plug-ins in a non-interfering manner. Formally proving this conjecture is currently under way and will be reported on in the future.
- Another important direction of work concerns the fact that our plug-ins are only supervisory. Supervision is, by definition, a restriction of the executions that the base application can generate. There is, however, great interest in being able to develop plug-ins that can also augment or add to overall system behaviors in a modular manner. The apparent modular nature of our base configurations suggests that the unfolding methods adopted in this paper might also be used to design plug-ins that augment a system's executions in a modular manner.
- While the Petri net is a useful low-level model for analysis purposes, its use is inconvenient for program specification. There is significant interest in our ability to integrate high-level modeling formalisms such as the Unified Modeling Language (UML) with our Petri net tools.
- The application of these methods for the runtime reconfiguration of distributed software represents an application of these methodologies that can have an enormous impact on the development of mobile Internet based software. Future work is definitely need to more fully explore the scalability of these methods for such applications.

This paper represents an initial attempt to assess the relevance of existing control theories to software engineering. In particular, it seems that software development is often an ad hoc process in which the user (rather than the designer) is responsible for assuring software reliability. As distributed software becomes increasingly important in the control and management of critical systems like the electric power grid or air traffic control, it is essential that these software systems be *engineered* in the same sense that we engineer planes, spacecraft, and other physical systems. In other words, our hope is that the methodologies presented in this paper provide a framework in which to formally engineer critical distributed object software with provable guarantees of program quality.

## References

1. P.J. Ramadge and W.M. Wonham, "Supervisory control of a class of discrete event processes", *SIAM Journal of Control and Optimization*, 25(1), pp. 206-230, 1987.

2. J. Engelfriet, “Branching processes of Petri nets”, *Acta Informatica*, 28, 575-591, 1991.
3. R. Laddaga (guest editor), special issue on “Robust software and self-adaptation” *IEEE Intelligent Systems and Their Applications*, Vol. 14, No. 3, May/June 1999.
4. J. Veitch and R. Laddaga (guest editors), special issue on distributed dynamic systems, Communications of the ACM, Vol 41, No. 5, May 1998.
5. J. Doyle, B. Francis, and A. Tannenbaum, Feedback Control Theory, MacMillan Press, 1992.
6. A. Giua, “Blocking and Controllability of Petri Nets in Supervisory Control”, *IEEE Trans. on Automatic Control*, Vol 39(4), 1994.
7. K. Yamalidou, J. Moody, M.D. Lemmon and P.J. Antsaklis, Feedback Control of Petri nets based on Place Invariants, *Automatica*, Vol. 32, No. 1, pp. 15-28, 1996.
8. J.O. Moody, P.J. Antsaklis, and M.D. Lemmon, “Application of Automatic Petri Net Control Design”, *proceedings of INRIA/IEEE Conference sur les technologies emergentes et l’automatisation de systemes de fabrication*, October 10-13, Paris, France.
9. K.X. He and M.D. Lemmon, ”On the transformation of liveness-enforcing marking based supervisors into monitor supervisors”, submitted to the IEEE Conference on Decision and Control, Sydney Australia, December 2000.
10. W.M.Wonham and P.J. Ramadge, “On the supremal controllable sublanguage of a given language”, *SIAM Journal of Control and Optimization*, 25(3), pp. 637-659, May 1987.
11. R.S. Sreenivas, “On supervisory policies that enforce liveness in complete controlled Petri nets with directed cut-places and cut-transitions”, *IEEE Trans. on Automatic Control*, Vol. 44(6), June 1999, pp. 1221-1225.
12. Y. Li and W.M. Wonham, “control of vector discrete-event systems: controller synthesis”, *IEEE Transactions on Automatic Control*, Vol. 39(3), pp. 512-530, 1994.
13. K.X. He and M.D. Lemmon, ”On the existence of liveness-enforcing supervisory policies of discrete-event systems modeled by *n*-safe Petri nets”, *Proceedings of IFAC conference on Control System Design*, Special issue on Petri nets, Slovakia, June 2000.
14. Vogler, W. (1992), *Modular Construction and Partial Order Semantics of Petri Nets*, Lecture Notes in Computer Science, Vol. 625, Springer-Verlag, 1992.
15. P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem.* PhD thesis, University of Liege, Computer Science Department, November 1994.
16. P. Godefroid and P. Wolper, ”Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties,” *Formal Methods in System Design*, Kluwer Academic Publishers, Vol. 2, No. 2, April 1993, pp. 149-164.
17. K. McMillan,”Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits”, *Computer Aided Verification, 4th International Workshop (CAV’92)*, (Bochmann and Probst (eds.), LLNCS Vol 663, Springer Verlag, 164-177, 1992.
18. A. Kondratyev, M. Kishinevsky, A. Taubing, and S. Ten, “Structural approach for the analysis of Petri nets by reduced unfoldings”, *Proceedings of the 17th International Conference on Application and Theory of Petri Nets*, Osaka Japan, June 24-28, 1996.
19. K.X. He and M.D. Lemmon, “Liveness verification of discrete event systems modeled by *n*-safe Petri nets”, to appear in Proceedings of the 21st International Conference on Application and Theory of Petri Nets, Denmark, June 2000.
20. J. Esparza, S. Romer, and W. Vogler, “An improvement of McMillan’s unfolding algorithm”, *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems*, (T. Magaria and B. Steffen, eds.), LNCS Vol. 1055, Springer-Verlag, 1996.

# Protocol Re-synthesis Based on Extended Petri Nets*

Khaled El-Fakih[1], Hirozumi Yamaguchi[2],
Gregor v. Bochmann[1], and Teruo Higashino[2]

[1] School of Information Technology and Engineering, University of Ottawa,
150 Louis Pasteur, Ottawa, Ontario K1N 6N5, Canada
{kelfakih,bochmann}@site.uottawa.ca
[2] Graduate School of Engineering Science, Osaka University,
1-3 Machikaneyamacho, Toyonaka, Osaka 560-8531, Japan
{h-yamagu,higashino}@ics.es.osaka-u.ac.jp

**Abstract.** Protocol synthesis is used to derive a specification of a distributed system from the specification of the services to be provided by the system to its users. Maintaining such a system involves applying frequent minor modifications to the service specification due to changes in the user requirements. In order to reduce the maintenance costs of such a system, we present an original method that consists of a set of rules that avoid complete protocol synthesis after these modifications. These rules are given for a system modeled as an extended Petri net. An application example is given along with some experimental results.

## 1 Introduction

Synthesis methods have been used (for surveys see [5, 6]) to derive a specification of a distributed system (hereafter called *protocol specification*) automatically from a given specification of the service to be provided by the distributed system to its users (called *service specification*). The service specification is written like a program of a centralized system, and does not contain any specification of the message exchange between different physical locations. However, the protocol specification contains the specification of communications between protocol entities (PE's) at the different locations.

A number of existing protocol synthesis strategies have been described in the literature. The first strategy, [9, 3, 4, 8, 10, 12, 14, 17, 18], aims at implementing complex control-flows using several computational models such as LOTOS, Petri nets, FSM/EFSM and temporal logic. The second strategy, [20, 23, 19, 24, 22], aims at satisfying the timing constraints specified by a given service specification in the derived protocol specification. This strategy deals with real-time distributed systems. The last strategy, [21, 25, 11, 15, 7, 16], deals with the management of distributed resources such as files and databases. The objective here,

* This work was partially funded by Communications and Information Technology Ontario (CITO).

is to determine how the values of these distributed resources are updated or exchanged between PE's for a given fixed resource allocation on different physical locations.

Some methods in the last strategy, especially these presented in our previous research work[26], have tried to synthesize a service specification by deriving its corresponding protocol specification with minimum communication costs and optimal allocation of resources.

As an example, we consider a Computer Supported Cooperative Work (CSCW) software development process. This process is distributed among engineers (developers, designers, managers and others). Each engineer has his own machine (PE) and participates in the development process using distributed resources (drafts, source codes, object codes, multimedia video and audio files, and others) placed on different machines. Considering the need for using these resources between different computers, we derive, using our protocol synthesis method, the engineer's sub-processes (protocol specification) knowing the whole software development cycle (service specification) and we decide on an allocation of resources that would minimize the communication costs. Both the service and protocol specifications are described using extended Petri nets.

In realistic applications, maintaining a system modeled by a given extended Petri net specification, involves modifying its specification as a result of changes in the user requirements. Synthesizing the whole system after each modification is considered expensive and time consuming. Therefore, it is important to re-synthesize the modified parts of service specification in order to reduce the maintenance cost, which was reported to account for as much as two-thirds of the cost of software production [30].

In this paper, we present a new method for re-synthesizing the protocol specification from a modified service specification. The method consists of a set of rules that would be applied to different PE's after a modification to the service specification, in order to produce new synthesized (henceforth called re-synthesized) PE's. The parts of the protocol specification that correspond to the unmodified parts of the service specification are preserved intact. As shown later, this method reduces the cost of synthesizing the whole system after each modification.

This paper is organized as follows. Section 2 gives examples of service and protocol specifications, and Section 3 describes the protocol synthesis method. Based on this method, we present in Section 4 protocol re-synthesis method along with some application examples in Section 5. Section 6 concludes this paper and includes our insights for future research.

## 2 Service Specification and Protocol Specification

### 2.1 Petri Net Model with Registers

We use an extended Petri net model called a *Petri Net with Registers* (*PNR* in short) [15] to describe both service and protocol specifications of a distributed

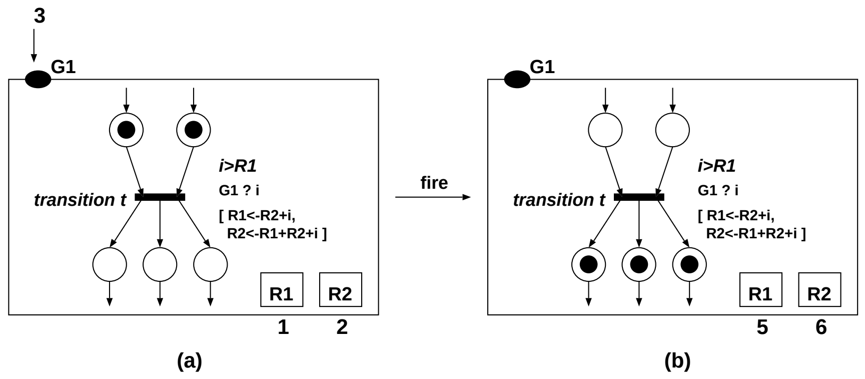


**Fig. 1.** Register Values and Token Locations before and after Firing of Transition in PNR

system. In this model, an I/O event between users and the system followed by the calculation of new values of variables inside the system is associated with the firing of a transition. Since distributed systems contain some variables (*e.g.* databases and files) and their values are updated according to inputs from users, they can be modeled by $PNR$ naturally.

Each transition $t$ in $PNR$ has a label $\langle \mathcal{C}(t), \mathcal{E}(t), \mathcal{S}(t) \rangle$, where $\mathcal{C}(t)$ is a pre-condition statement (one of the firing conditions of $t$), $\mathcal{E}(t)$ is an event expression (which represents I/O) and $\mathcal{S}(t)$ is a set of substitution statements (which represents parallel updates of data values). Consider, for example, transition $t$ of Fig. 1 where $\mathcal{C}(t) =$"$i > R_1$", $\mathcal{E}(t) =$"$G_1?i$" and $\mathcal{S}(t) =$"$R_1 \leftarrow R_2+i, R_2 \leftarrow R_1+R_2+i$". $i$ is an input variable, which keeps an input value and its value is referred by only the transition $t$. $R_1$ and $R_2$ are registers, which keep assigned values until new values are assigned, and their values may be referred and updated by all the transitions in $PNR$ (that is, global variables). $G_1$ is a gate, a service access point (interaction point) between users and the system. Note that "?" in $\mathcal{E}(t)$ means that $\mathcal{E}(t)$ is an input event.

A transition may fire if (a) each of its input place has one token, (b) the value of $\mathcal{C}(t)$ is true and (c) an input value is given through the gate in $\mathcal{E}(t)$ (if $\mathcal{E}(t)$ is an input event). Assume that an integer of value 3 has been given through gate $G_1$, and the current values of registers $R_1$ and $R_2$ are 1 and 2, respectively. In this case the value of "$i > R_1$" is true and the transition may fire. If it fires, the event "$G_1?i$" is executed and the input value 3 is assigned to input variable $i$. Then "$R_1 \leftarrow R_2+i$" and "$R_2 \leftarrow R_1+R_2+i$" are executed in parallel. Therefore after the firing, the tokens are moved and the values of registers $R_1$ and $R_2$ are changed to five ($= 2 + 3$) and six ($= 1 + 2 + 3$), respectively (Fig. 1(b)).

Formally, $\mathcal{E}(t)$ is one of the following three events: "$G_s$ $!exp$", "$G_s$ $?iv$", or "$\tau$". "$G_s$ $!exp$" is an output event and it means that the value of expression "$exp$", whose arguments are registers, is output through gate $G_s$. "$G_s$ $?iv$" is

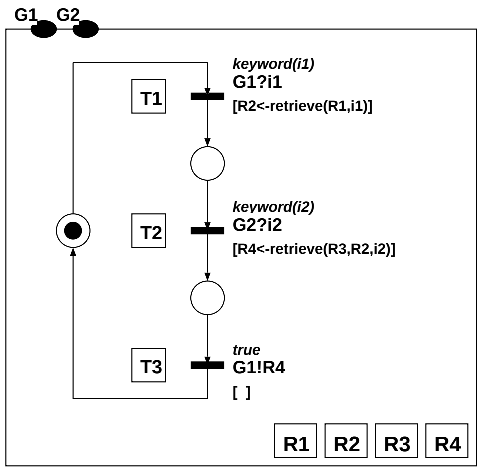


**Fig. 2.** Service Specification

an input event and it means that the value given through $G_s$ is assigned to the input variable "$iv$". "$\tau$" is an internal event, which is unobservable from the users. $\mathcal{S}(t)$ is a set of substitution statements, each of the form "$R_w \leftarrow exp_w$", where $R_w$ is a register and $exp_w$ is an expression whose arguments are from the input variable in $\mathcal{E}(t)$ and registers. If $t$ fires, $\mathcal{E}(t)$ is executed followed by the parallel execution of statements in $\mathcal{S}(t)$.

## 2.2 Service Specification

At a highly abstracted level, a distributed system is regarded as a centralized system which works and provides services as a single "virtual" machine. The number of actual PE's and communication channels among them are hidden. The specification of the distributed system at this level is called a *service specification* and denoted by $Sspec$.

Actual resources of a distributed system may be located on some physical machines, called protocol entities. However, only one virtual machine is assumed at this level. Fig. 2 shows $Sspec$ of a simple database system which has only three transitions. The system receives a keyword (input variable $i_1$) through gate $G_1$, retrieves an entry corresponding to the keyword from a database (register $R_1$), and stores the result to register $R_2$. This is done on transition $T_1$. Then the system receives another keyword (input variable $i_2$) through gate $G_2$, retrieves an entry corresponding to the keyword and the retrieved entry (register $R_2$) from another database (register $R_3$), and stores the result to register $R_4$. This is done on transition $T_2$. Finally the system outputs the second result (the value of register $R_4$) through $G_1$ on transition $T_3$ and returns to the initial state.

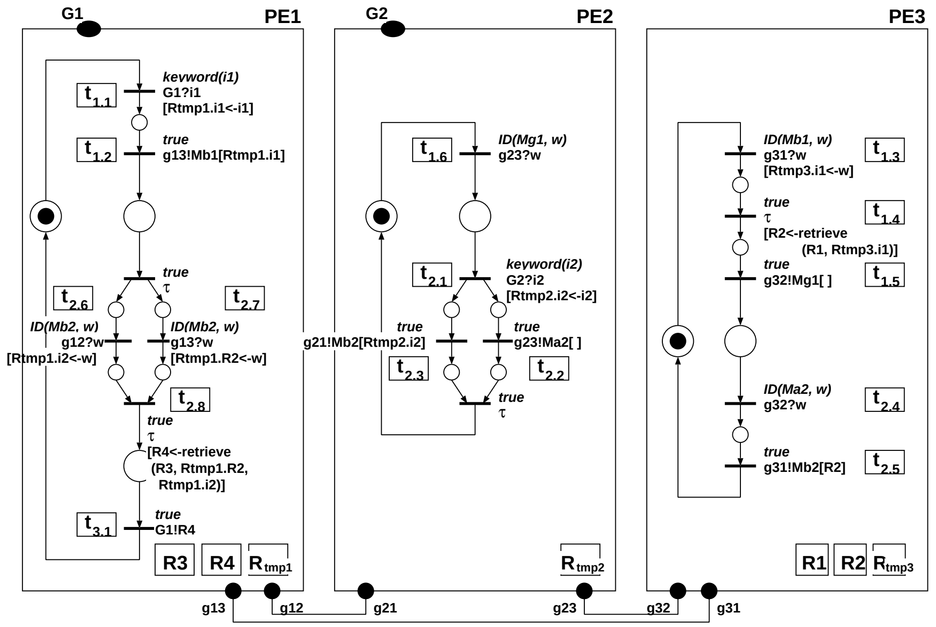


**Fig. 3.** Protocol Specification

## 2.3 Protocol Specification

A distributed system is a communication system which consists of $p$ protocol entities $PE_1$, $PE_2$, ... and $PE_p$. We assume a duplex and reliable communication channel with infinite capacity buffers at both ends, between any pair of $PE_i$ and $PE_j$. The $PE_i$ ($PE_j$) side of the communication channel is represented as gate $g_{ij}$ ($g_{ji}$). Moreover, we assume that some resources (registers and gates) are allocated to certain PE's of the distributed system.

Two PE's communicate with each other by exchanging messages. If $PE_i$ executes an output event "$g_{ij}!M[R_w]$", the value of register $R_w$ located on $PE_i$ is sent to $PE_j$ through the communication channel between them and put into the buffer at $PE_j$'s end. $M$ is an identifier to distinguish several values which may exist at the same time on the same channel. $PE_j$ can take the value identified by $M$ from the buffer, by executing an input event "$g_{ji}?w$" with a pre-condition $ID(M, w)$. $ID(M, w)$ is a predicate whose value is true *iff* the identifier in input variable $w$ is $M$. Note that more than one register's or input variable's value can be sent at a time. If a received data contains multiple values, they are distinguished by suffix such as $w.R_1$ and $w.i$. A set of an identifier and register/input values is called a message. A message may contain no value and sending such a message is represented as an output event "$g_{ij}!M[\ ]$".

In order to implement a distributed system which consists of $p$ PE's, we must specify the behavior of these PE's. A specification of $PE_k$ is called a *protocol entity specification* and denoted by $Pspec_k$. A set of $p$ protocol entity specifications $\langle$ $Pspec_1$, ..., $Pspec_p$ $\rangle$ is called a *protocol specification* and denoted by $Pspec^{\langle 1,p \rangle}$. We need a protocol specification to implement the distributed system.

As an example, let us assume that there are three PE's $PE_1$, $PE_2$ and $PE_3$ in order to implement the service specification of Fig. 2. We also assume that an allocation of resources to these PE's has been fixed as follows. $PE_1$ has the gate $G_1$ and the registers $R_3$ and $R_4$, $PE_2$ has the gate $G_2$, and $PE_3$ has the registers $R_1$ and $R_2$. Note that in addition to these registers, we assume that each $PE_i$ has another register $Rtmp_i$ to keep received values given through gates (inputs and message contents). $Rtmp_i$ can contain several values. The values can be distinguished by adding the name of the value as suffix, such as $Rtmp_1.R_3$[1]. Fig. 3 shows an example of $Pspec^{\langle 1,3 \rangle}$, which provides the service of Fig. 2, based on this allocation of resources.

According to the specification of Fig. 3, $PE_1$ first receives an input (input variable $i_1$) through $G_1$ and stores it to $Rtmp_1.i_1$ (on transition $t_{1.1}$). Then it sends the value of $Rtmp_1.i_1$ to $PE_3$ as a message (on transition $t_{1.2}$), since $PE_3$ needs the value of $i_1$ to change the value of $R_2$. $PE_3$ receives and stores the value to $Rtmp_3.i_1$ on transition $t_{1.3}$. Then it changes the value of $R_2$ using its own value and the value of $Rtmp_3.i_1$ on transition $t_{1.4}$, and sends a message to $PE_2$ on transition $t_{1.5}$. When $PE_2$ receives the message on transition $t_{1.6}$, $PE_2$ knows that it can now check the value of $\mathcal{C}(T_2)$ and execute $\mathcal{E}(T_2)$. $PE_2$ receives an input (input variable $i_2$) and stores it to $Rtmp_2.i_2$ on transition $t_{2.1}$, and sends two messages. One is to send the value of $i_2$ to $PE_1$ (on transition $t_{2.3}$) and another is to incite $PE_3$ to send the value of $R_2$ to $PE_1$ (on transition $t_{2.2}$). $PE_1$ receives these values and stores them to $Rtmp_1.i_2$ and $Rtmp_1.R_2$ on transitions $t_{2.6}$ and $t_{2.7}$, respectively. Then it changes the value of $R_4$ on transition $t_{2.8}$. Finally, $PE_1$ outputs the value of $R_4$ on transition $t_{3.1}$ and $PE_1$, $PE_2$ and $PE_3$ return to their initial states.

As exemplified in the above discussion, PE's cooperate with each other by exchanging messages. The communication between different PE's may be quite complex and it is difficult to design protocols that behave correctly. Therefore we would like to derive a protocol specification automatically, such that it provides the same service as a given service specification.

## 3 Synthesis Overview

A method for deriving protocol specification with an optimal allocation of resources from a given service specification is presented in this section. This method is based on a set of rules (called henceforth synthesis rules) that specify how to execute each transition $T_x = \langle \mathcal{C}(T_x), \mathcal{E}(T_x), \mathcal{S}(T_x) \rangle$ of the service specification by the corresponding PE's in the protocol specification. Furthermore, based on

[1] We can realize such a register that contains several values by using several registers. However, for simplicity of discussion, we use these registers.

these rules, it decides on an optimal allocation of resources (registers and gates) amongst different derived PE's.

### 3.1 Synthesis Rules

For executing a transition $T_x = \langle \mathcal{C}(T_x), \mathcal{E}(T_x), \mathcal{S}(T_x) \rangle$ of the service specification by the corresponding set of transitions $t_{x.1}, t_{x.2}, \dots$ of the PE's in the protocol specification, we proceed as follows.

- The PE that has gate $G_s$ used in $\mathcal{E}(T_x)$ (say PEstart$(T_x)$) checks the value of $\mathcal{C}(T_x)$ (pre-condition statement) and executes $\mathcal{E}(T_x)$ (event expression).
- After that, the PE sends messages called $\alpha$-messages to the PE's which have the registers used in the arguments of $\mathcal{S}(T_x)$ (substitution statements).
- In response, these PE's send the register values to the PE's which have the registers to be updated in $\mathcal{S}(T_x)$ (PEsubst$(T_x)$ denotes the set of those PE's) as messages called $\beta$-messages.
- The substitution statements are executed and notification messages called $\gamma$-messages are sent to those PE's which will start the execution of the next transitions.

For example, for transition $T_2$ of the service specification in Fig. 2, PEstart$(T_2)$ is PE$_2$ and PEsubst$(T_2)$ is $\{$PE$_1\}$. PE$_2$ checks the value of pre-condition statement "*keyword*$(i_2)$" and executes "$G_2?i_2$" on transition $t_{2.1}$. Then PE$_2$ sends an $\alpha$-message "$Ma_2$" to PE$_3$ on transition $t_{2.2}$ since PE$_3$ has register $R_2$ which is used to substitute the value of $R_4$. PE$_2$ also sends the input value to PE$_1$ as a $\beta$-message "$Mb_2$" on transition $t_{2.3}$. PE$_3$ receives the $\alpha$-message "$Ma_2$" on transition $t_{2.4}$ and sends the value of $R_2$ to PE$_1$ as a $\beta$-message "$Mb_2$" on transition $t_{2.5}$. PE$_1$ receives these two $\beta$-messages on transitions $t_{2.6}$ and $t_{2.7}$, and then executes "$R_4 \leftarrow$ retrieve$(R_3, R_2, i_2)$" on transition $t_{2.8}$ using its own register $R_3$ and the received values of $R_2$ and $i_2$. The PE's which will start the execution of next transition $T_3$ is $PE_1$ itself. Therefore, PE$_1$ does not send any $\gamma$-message. Then PE$_1$ starts the execution of $T_3$ on transition $t_{3.1}$.

In Fig. 4, we present the details of the above rules [26], that are classified into action and message rules. Action rules specify which PE checks the pre-condition and executes the event and substitution statements of $T_x$. Message rules specify how the PE's exchange messages, and the contents and types of these messages.

Three types of messages are exchanged for the execution of $T_x$. (1) $\alpha$-messages are sent by the PE that starts the execution of $T_x$ (*i.e.* PEstart$(T_x)$) to inform those PE's who need to send their registers' values to other PE's, that they can go ahead and send these values. Thus, an $\alpha$-message does not contain values of registers. (2) $\beta$-messages are sent in order to let each PE which executes some substitution statements of $T_x$ (*i.e.* PE$_k \in$PEsubst$(T_x)$) know the timing and some values of registers' it needs for executing these statements. (3) $\gamma$-messages are sent to each PE$_m \in$PEstart$(T_x \bullet\bullet)$, note that $T_x \bullet\bullet$ is the set of each next transition of $T_x$, to let it know the timing and some values of registers it needs to start executing the next transitions (*i.e.* transitions in $T_x \bullet\bullet$).

We let $T_x = \langle \mathcal{C}(T_x), \mathcal{E}(T_x), \mathcal{S}(T_x) \rangle$ be a transition of $Sspec$.

**[Action Rules]**

($\mathrm{A}_1$) The PE which has the gate appearing in $\mathcal{E}(T_x)$ (denoted by $G_s$) checks that
(a) the value of $\mathcal{C}(T_x)$ is true,
(b) the execution of the previous transitions of $T_x$ has been finished and
(c) an input has been given through $G_s$ if $\mathcal{E}(T_x)$ is an input event.
Then the PE executes $\mathcal{E}(T_x)$. This PE is denoted by PEstart($T_x$).

($\mathrm{A}_2$) After ($\mathrm{A}_1$), the PE's which have at least one register whose value is changed in the substitution statements $\mathcal{S}(T_x)$ execute the corresponding statements in $\mathcal{S}(T_x)$. The set of these PE's is denoted by PEsubst($T_x$).

**[Message Rules]**

($\mathrm{M}_{\beta 1}$) Each $\mathrm{PE}_k \in$PEsubst($T_x$) must receive at least one $\beta$-message from some PE's (each called $\mathrm{PE}_j$) in order to know the timing and values of registers it needs for executing its substitution statements (see ($\mathrm{M}_{\beta 2}$)), except where $\mathrm{PE}_k$=PEstart($T_x$), in this case $\mathrm{PE}_k$ already knows the timing to start executing its substitution statements of $T_x$.

($\mathrm{M}_{\beta 2}$) If $\mathrm{PE}_k \in$PEsubst($T_x$) needs the value of some register (say $R_z$) in order to execute its substitution statements, then $\mathrm{PE}_k$ must receive $R_z$ through a $\beta$-message if $R_z$ is not in $\mathrm{PE}_k$.

($\mathrm{M}_{\beta 3}$) Each $\mathrm{PE}_j$ that sends some values of registers to $\mathrm{PE}_k \in$PEsubst($T_x$) through a $\beta$-message, knows the timing to send these values by receiving an $\alpha$-message from PEstart($T_x$). Note, if $\mathrm{PE}_j$=PEstart($T_x$) then $\mathrm{PE}_j$ knows the timing to send these values without receiving an $\alpha$-message.

($\mathrm{M}_{\alpha}$) After ($\mathrm{A}_1$), the only PE that can send $\alpha$-messages to the PE's which need them is PEstart($T_x$).

($\mathrm{M}_{\gamma 1}$) Each $\mathrm{PE}_m \in$PEstart($T_x \bullet\bullet$), where $T_x \bullet\bullet$ is the set of next transitions of $T_x$, must receive a $\gamma$-message from each $\mathrm{PE}_k \in$PEsubst($T_x$) after ($\mathrm{A}_2$), except where $m = k$. This allows $\mathrm{PE}_m$ to know that the execution of the substitution statements of $T_x$ had been finished.

($\mathrm{M}_{\gamma 2}$) Each $\mathrm{PE}_m \in$PEstart($T_x \bullet\bullet$) must receive at least one $\gamma$-message from some $\mathrm{PE}_l$ (where $m \neq l$) in order to know that the execution of $T_x$ had been finished and/or to know some values of registers it needs to evaluate and execute its condition and event expression, respectively.

($\mathrm{M}_{\gamma 3}$) Each $\mathrm{PE}_l$ that sends a $\gamma$-message to $\mathrm{PE}_m \in$PEstart($T_x \bullet\bullet$) :
(a) must be in PEsubst($T_x$) (see ($\mathrm{M}_{\gamma 1}$)), or
(b) must receive an $\alpha$-message from PEstart($T_x$) to know the timing to send the $\gamma$-message to $\mathrm{PE}_m$, or
(c) it is itself PEstart($T_x$). In this case, $\mathrm{PE}_l$ sends the $\gamma$-message to let $\mathrm{PE}_m$ know the timing and/or some values of registers to start evaluating and executing its condition and event expressions.

($\mathrm{M}_{\gamma 4}$) If $\mathrm{PE}_m \in$PEstart($T_x \bullet\bullet$) needs the value of some register (say $R_v$) in order to evaluate and/or execute its substitution statements, then $\mathrm{PE}_m$ must receive $R_v$ through a $\gamma$-message if $R_z$ is not in $\mathrm{PE}_m$.

**Fig. 4.** Derivation Method in Detail

### 3.2 Integer Linear Programming Model for Protocol Derivation

Based on the above synthesis rules, we determine a behavior of the derived PE's that would minimize their communication cost while optimally allocating their resources, using an Integer Linear Programming (ILP) model. This cost could be based on the number of messages to be exchanged between different PE's [25]. Moreover, other cost criteria can also be considered such as the costs of resource allocation, size of messages exchanged between different PE's, and frequencies of transition execution.

The ILP Model (for details see [26, 25]) consists of an objective function that minimizes the communication cost and decides on an optimal allocation of resources, based on a set of constraints. These constraints are based on the above synthesis rules, and they consist of 0-1 integer variables indicating (a) whether a PE should send a message or not, (b) whether a message contains a register value or not, or (c) whether a register/gate is allocated to a PE or not.

## 4 Protocol Re-synthesis

In this section, we present our new method for re-synthesizing the protocol specification from a modified service specification. The method consists of a set of rules that would be applied to different PE's after a modification to the service specification, in order to produce new synthesized (re-synthesized) PE's.

For each simple modification (henceforth called *atomic modification*) made on the service specification $Sspec$, we define its corresponding *atomic re-synthesis rules*. As shown later, these atomic re-synthesis rules can also be sequentially applied to deal with more than one modification. Note that the atomic re-synthesis rules are based on the synthesis rules described in Section 3. Consequently, we show next to the description of each re-synthesis rule its corresponding synthesis rule.

### 4.1 Atomic Modifications and Their Corresponding Re-synthesis Rules

For each of the following possible atomic modifications to $Sspec$, we present its corresponding atomic re-synthesis rules. Note that each modification to $Sspec$ changes the label of a transition $T_x$ in $Sspec$ from $\langle\mathcal{E}(T_x), \mathcal{C}(T_x), \mathcal{S}(T_x)\rangle$ to $\langle\mathcal{E}'(T_x), \mathcal{C}'(T_x), \mathcal{S}'(T_x)\rangle$. For convenience, we denote the following sets of registers:

- $Rev^x$: the set of registers that PEstart($T_x$) needs to evaluate $\mathcal{C}(T_x)$ or execute $\mathcal{E}(T_x)$
- $Rrsub_i^x$: the set of registers that are used in PE$_i$ $\in$PEsubst($T_x$) to execute the statements in $\mathcal{S}(T_x)$
- $Rcsub_i^x$: the set of registers that are defined (*i.e.* referenced) by the left-hand-sides of the substitution statements in $\mathcal{S}(T_x)$ in PE$_i$ $\in$PEsubst($T_x$).

[**Atomic Modifications**]

1. $Rev^x \leftarrow Rev^x \setminus \{R_h\}$
2. $Rev^x \leftarrow Rev^x \cup \{R_h\}$
3. $Rrsub_k^x \leftarrow Rrsub_k^x \setminus \{R_h\}$
4. $Rrsub_k^x \leftarrow Rrsub_k^x \cup \{R_h\}$
5. $Rcsub_k^x \leftarrow Rcsub_k^x \setminus \{R_h\}$
6. $Rcsub_k^x \leftarrow Rcsub_k^x \cup \{R_h\}$

[**Atomic Re-synthesis Rules**]

1. $Rev^x \leftarrow Rev^x \setminus \{R_h\}$:
   The following rules take into account that the value of $R_h$ which has been sent to PEstart($T_x$) is no longer necessary after the modification. These rules are applied to the part of the protocol specification where each previous transition (say $T_w$) of $T_x$ is executed, if applicable.
   (a) Each PE (say PE$_l$) which sends a $\gamma$-message including the value of $R_h$ to PEstart($T_x$), should exclude the value of $R_h$ from the $\gamma$-message (*c.f.* **synthesis rule (M$_{\gamma4}$)**).
   (b) If (a) is done, then the $\gamma$-message can be deleted only if
      – PE$_l$ $\notin$PEsubst($T_w$) (*c.f.* **synthesis rule (M$_{\gamma1}$)**),
      – there is still at least one $\gamma$-message sent to PEstart($T_x$) after deleting it (*c.f.* **synthesis rule (M$_{\gamma2}$)**) and
      – it no longer has values (*c.f.* **synthesis rule (M$_{\gamma4}$)**).
   (c) If (b) is done, then an $\alpha$-message sent to PE$_l$ can be deleted only if PE$_l$ no longer sends $\beta$- and $\gamma$-messages (*c.f.* **synthesis rule (M$_{\gamma3}$)(b)**).
2. $Rev^x \leftarrow Rev^x \cup \{R_h\}$:
   The following rules take into account that the value of $R_h$ must be sent to PEstart($T_x$) after the modification. These rules are applied to the part of the protocol specification where each previous transition (say $T_w$) of $T_x$ is executed, if applicable.
   (a) One of the PE's which have $R_h$ and send $\gamma$-messages to PEstart($T_x$) should include the value of $R_h$ in its $\gamma$-message to PEstart($T_x$), if such a PE exists (*c.f.* **synthesis rule (M$_{\gamma4}$)**).
   (b) Otherwise, one of the PE's which have $R_h$ should send a new $\gamma$-message which includes the value of $R_h$ to PEstart($T_x$). If the PE does not receive $\alpha$-messages and is not PEstart($T_x$), PEstart($T_w$) should send an $\alpha$-message to the PE. (*c.f.* **synthesis rule (M$_{\gamma3}$)**).
3. $Rrsub_k^x \leftarrow Rrsub_k^x \setminus \{R_h\}$:
   The following rules take into account that the value of $R_h$ sent to PE$_k$ is no longer necessary after the modification. These rules are applied to the part of the protocol specification where $T_x$ is executed.
   (a) Each PE (say PE$_j$) which sends a $\beta$-message including the value of $R_h$ to PE$_k$ should exclude the value from the $\beta$-message (*c.f.* **synthesis rule (M$_{\beta2}$)**).
   (b) If (a) is done, then the $\beta$-message can be deleted only if

   - there is still at least one $\beta$-message sent to PE$_k$ after deleting it (*c.f.* **synthesis rule (M$_{\beta 1}$)**) and
   - it no longer has values (*c.f.* **synthesis rule (M$_{\beta 2}$)**).

   (c) If (b) is done, the $\alpha$-message sent to PE$_j$ can be deleted only if PE$_j$ no longer sends $\beta$- and $\gamma$-messages. (*c.f.* **synthesis rule (M$_{\beta 3}$)**).

4. $Rrsub_k^x \leftarrow Rrsub_k^x \cup \{R_h\}$:
   The following rules take into account that the value of $R_h$ must be sent to PE$_k$ after the modification. These rules are applied to the part of the protocol specification where $T_x$ is executed.
   (a) One of the PE's which have $R_h$ and send $\beta$-messages to PE$_k$ should include the value of $R_h$ to its $\beta$-message to PE$_k$, if such a PE exists. (*c.f.* **synthesis rule (M$_{\beta 2}$)**).
   (b) Otherwise, one of PE's which have $R_h$ should send a new $\beta$-message which includes $R_h$ to PE$_k$. If the PE does not receive $\alpha$-messages and is not PEstart($T_x$), PEstart($T_x$) should send an $\alpha$-message to the PE.
5. $Rcsub_k^x \leftarrow Rcsub_k^x \setminus \{R_h\}$:
   Removing a substitution statement. Usually, this may cause an additional modification $Rrsub_k^x \leftarrow Rrsub_k^x \setminus \{R_{h_1}, R_{h_2}, ..., R_{h_k}\}$, since the deleted statement uses values of registers. In this case, we consider that the atomic modification (3) was made on *Sspec* $k$ times and apply its corresponding atomic re-synthesis rule (3) $k$ times.
6. $Rcsub_k^x \leftarrow Rcsub_k^x \cup \{R_h\}$:
   Adding a substitution statement. Usually, this may cause an additional modification $Rrsub_k^x \leftarrow Rrsub_k^x \cup \{R_{h_1}, R_{h_2}, ..., R_{h_k}\}$, since the added statement uses values of registers. As the case of the re-synthesis rule (5), we apply the atomic re-synthesis rule (4) $k$ times.

## 4.2 Modifications to the Service Specification

In this section, we describe how modifications to *Sspec* can be represented as the set of atomic modifications presented in the previous subsection. We consider modifications to the label of a transition $T_x$ of *Sspec*.

- If $\mathcal{E}(T_x)$ (or $\mathcal{C}(T_x)$) is modified to $\mathcal{E}'(T_x)$ (or $\mathcal{C}'(T_x)$), then this modification can be represented as a set of the atomic modifications of type (1) and/or (2) which involve adding and/or removing registers from the set of registers $Rev^x$ that PEstart($T_x$) needs to execute $\mathcal{E}(T_x)$ (or evaluate $\mathcal{C}(T_x)$).
- If $\mathcal{S}(T_x)$ is modified to $\mathcal{S}'(T_x)$, then this modification can be represented by a sequence of atomic modifications of type (3), (4), (5) or (6), respectively.

## 4.3 Changing the Resource Allocation for the Protocol Specification

In some application areas, the allocation of resources between different PE's is necessary. For example, in distributed databases, adding a copy of an existing register to some PE's is necessary to increase the fault tolerance and balance the load amongst these PE's. Here we consider the case where a copy of an existing register $R_h$ in PE$_j$ is added to another PE PE$_k$. For each transition $T_x$ where the value of $R_h$ is changed (defined) in the substitution statement $\mathcal{S}(T_x)$, PE$_k$ must execute this substitution statement to update the value of register $R_h$. Consequently, this modification can be represented by the atomic modification (6).

## 5 Example and Experimental Results

### 5.1 Modeling the ISPW-6 Example

Protocol synthesis methods have been applied to many applications such as communication protocols, factory manufacturing systems[14], distributed cooperative work management[13] and so on.

In this section, we apply our synthesis method [26] to the distributed development of software that involves five engineers (project manager, quality assurance, design, and two software engineers). Each engineer has his own machine connected with the others, and participates in the development through a gate (interfaces) of this machine, using distributed resources placed on this machine. This distributed development process includes scheduling and assigning tasks, design modification, design review, code modification, test plans modification, modification of unit test packages, unit testing, and progress monitoring tasks. The engineers cooperate with each other to finish these sub-sequential tasks. The reader may refer to ISPW-6 [28] for a complete description of this process, which was provided as an example to help the understanding and comparison of various approaches to process modeling.

Figure 5 shows a workflow model of the above development process using *PNR*, where the engineers and resources needed to accomplish the tasks are indicated. We note that for convenience, we do not show the progress monitoring process tasks in Fig. 5.

We regard this workflow as the service specification, and we derive its corresponding protocol specification using the method and programs used in our previous work[26], where we have developed two programs that generate for the given specification its corresponding ILP problem constraints, and derive the protocol specifications using the synthesis rules. The tool lp_solve[29] is used to solve the ILP problem and obtain the minimal number of messages to be exchanged between the derived protocol entities. It took 639 seconds on MMX-Pentium 200MHz PC to synthesize the given specification. The optimal allocation of the registers is shown in Table 1 and the minimum number of messages to be exchanged between the different PE's is 40.

### 5.2 Experimental Results

In this section, we show the effectiveness of our re-synthesis method by comparing the time it takes to synthesize the given service specification again after an assumed modification to the time it takes using our re-synthesis method.

We consider the following modifications to the given service specification:

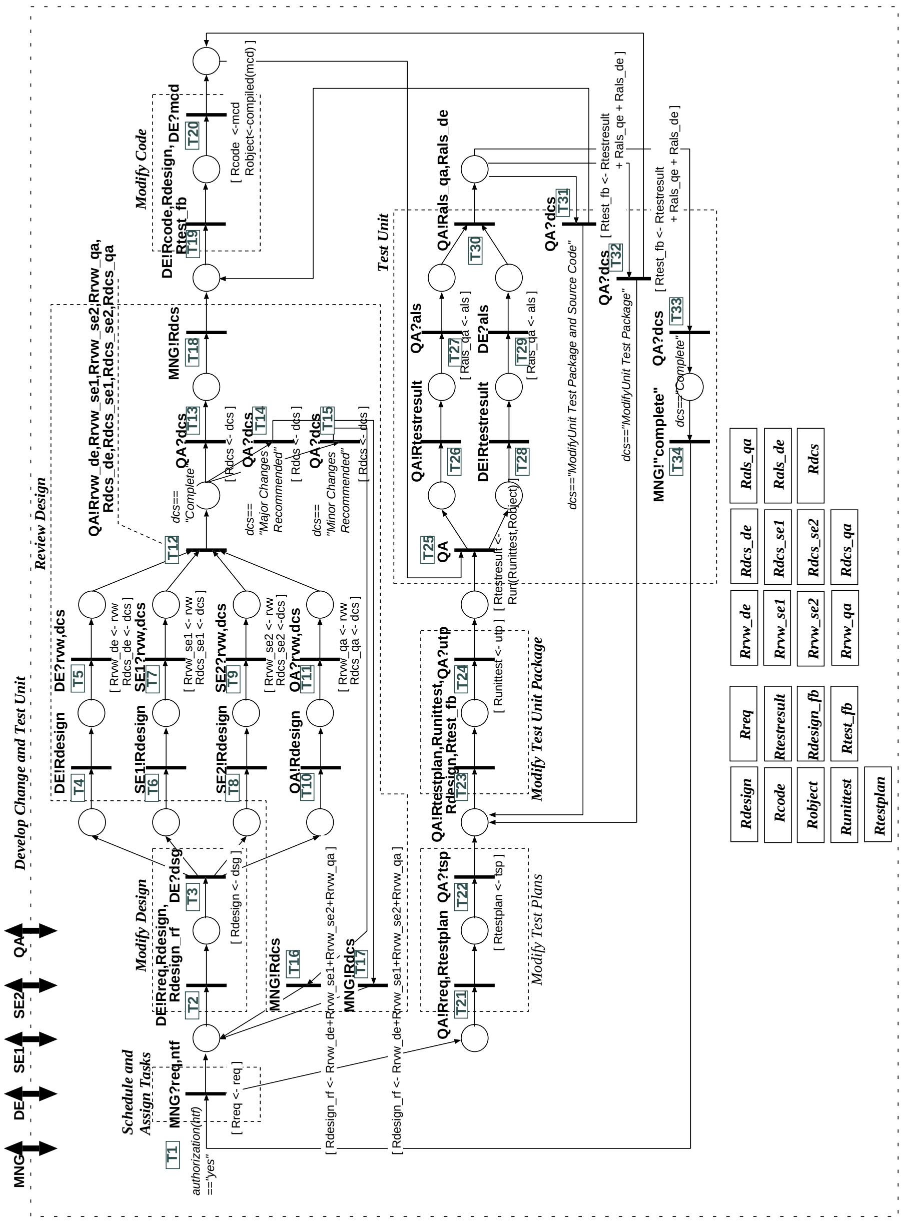


**Fig. 5.** Modeling the Core Problem in the ISPW-6 Example

|  | PE$_{mng}$ | PE$_{de}$ | PE$_{se1}$ | PE$_{se2}$ | PE$_{qa}$ |
|---|---|---|---|---|---|
| Gate | $MNG$ | $DE$ | $SE1$ | $SE2$ | $QA$ |
| Register |  | $R_{req}$<br>$R_{design}$<br>$R_{design\_fb}$<br>$R_{rvw\_de}$<br>$R_{dcs\_de}$<br>$R_{code}$<br>$R_{test\_fb}$<br>$R_{testplan}$ | $R_{rvw\_se1}$<br>$R_{dcs\_se1}$<br>$R_{unittest}$<br>$R_{testresult}$ | $R_{rvw\_se2}$<br>$R_{dcs\_se2}$ | $R_{rvw\_qa}$<br>$R_{dcs\_qa}$<br>$R_{dcs}$<br>$R_{object}$<br>$R_{alc\_qa}$<br>$R_{als\_de}$ |

**Table 1.** Optimal Allocation of Resources for Engineers' Machines

|  | Synthesis Time (sec.) |  | Number of Messages |  |
|---|---|---|---|---|
|  | Re-synthesis | Complete Synthesis | Re-synthesis | Complete Synthesis |
| case1 | 1 | 958 | 44 | 44 |
| case2 | 1 | 1021 | 46 | 46 |
| case3 | 1 | 940 | 40 | 40 |
| case4 | 1 | 1640 | 42 | 42 |

`MMX-Pentium 200 MHz, 128MB Memory`

**Table 2.** Experimental Results

1. An additional source code (register $R_{code\_new}$) is placed on the machine of the software engineer 1 (SE1), and the design engineer (DE) modifies and compiles it as well as $R_{code}$, in "Modify Code" (transitions $T_{19}$ and $T_{20}$).
2. An additional new unit test (register $R_{unittest\_new}$) is placed on the machine of the software engineer 2 (SE2), and the QA engineer (QA) modifies it as well as $R_{unittest}$, in "Modify Test Unit Package" ($T_{23}$ and $T_{24}$). Moreover, an additional test is done using the unit test in "Test Unit" ($T_{25}$).
3. DE analyzes the test feedback (register $R_{test\_fb}$) and gives his comments to QA. For this purpose, a new register $R_{report}$ is introduced on DE's machine and his comments are stored on it in transition $T_{20}$. Then it is shown to QA on $T_{25}$.
4. For fault tolerance, a new copy of the existing code $R_{code}$ (placed on PE$_{de}$) is placed on PE$_{mng}$.

After each modification, we have used the programs developed in [26] to measure the time (in seconds) it takes to synthesize the given specification. Moreover, we have also measured the time it took to re-derive the protocol specifications using the re-synthesis rules and a program that we have developed for this purpose. Table 2 shows these times. The reader can clearly see that the re-synthesize time is much less than the time for a complete synthesis. This is mainly due to the fact that by using the re-synthesis rules, we do not have to re-derive the whole protocol specifications after each modification. Moreover, we do not have to re-optimize the number of messages sent between different PE's because (as shown in Table 2) the re-derived protocol specifications still have optimal (or near-optimal in general cases) solutions.

## 6 Conclusion and Further Research

Based on our previous work on protocol synthesis of systems modeled as extended Petri nets, we have developed a set of rules that avoid complete synthesis after incremental modifications to such a system. These rules are applied to the affected parts of derived protocol specification. This would make protocol synthesis and maintenance more practical for realistic applications.

Currently, we are developing a re-synthesis method to specifications modeled as finite state machines. Moreover, we are investigating the extension of our re-synthesis method to specifications modeled as timed Petri nets.

## References

1. T. Murata, "Petri Nets: Properties, Analysis and Applications," *Proc. of the IEEE*, Vol. 77, No. 4, pp. 541–580, 1989.
2. R. Milner, "Communication and Concurrency," *Prentice-Hall*, 1989.
3. V. Carchiolo, A. Faro and D. Giordano, "Formal Description Techniques and Automated Protocol Synthesis," *Journal of Information and Software Technology*, Vol. 34, No. 8, pp. 513–421, 1992.
4. H. Erdogmus and R. Johnston, "On the Specification and Synthesis of Communicating Processes," *IEEE Trans. on Software Engineering*, Vol. SE-16, No. 12, 1990.
5. R. Probert and K. Saleh, "Synthesis of Communication Protocols: Survey and Assessment," *IEEE Trans. on Computers*, Vol. 40, No. 4, pp. 468–476, 1991.
6. K. Saleh, "Synthesis of Communication Protocols: an Annotated Bibliography," *ACM SIGCOMM Computer Communication Review*, Vol. 26, No. 5, pp. 40–59, 1996.
7. R. Gotzhein and G. v. Bochmann, "Deriving Protocol Specifications from Service Specifications Including Parameters," *ACM Trans. on Computer Systems*, Vol. 8, No. 4, pp. 255–283, 1990.
8. R. Langerak, "Decomposition of Functionality; a Correctness-Preserving LOTOS Transformation," *Proc. of 10th IFIP WG6.1 Symp. on Protocol Specification, Testing and Verification (PSTV-10)*, pp. 229–242, 1990.
9. C. Kant, T. Higashino and G. v. Bochmann, "Deriving Protocol Specifications from Service Specifications Written in LOTOS," *Distributed Computing*, Vol. 10, No. 1, pp. 29–47, 1996.
10. P. -Y. M. Chu and M. T. Liu, "Protocol Synthesis in a State-transition Model," *Proc. of COMPSAC '88*, pp. 505–512, 1988.
11. T. Higashino, K. Okano, H. Imajo and K. Taniguchi, "Deriving Protocol Specifications from Service Specifications in Extended FSM Models," *Proc. of 13th Int. Conf. on Distributed Computing Systems (ICDCS-13)*, pp. 141–148, 1993.
12. M. Nakamura, Y. Kakuda and T. Kikuno, "Component-based Protocol Synthesis from Service Specifications," *Computer Communications Journal*, Vol. 19, No. 14, pp.1200-1215, Dec. 1996.

13. K. Yasumoto, T. Higashino and K. Taniguchi, “Software Process Description Using LOTOS and its Enaction,” Proc. of the 16th Int. Conf. on Software Engineering (ICSE-16), pp. 169-179, 1994.
14. D. Y. Chao and D. T. Wang, “A Synthesis Technique of General Petri Nets,” *Journal of System Integration*, Vol. 4, pp. 67–102, 1994.
15. H. Yamaguchi, K. Okano, T. Higashino and K. Taniguchi, “Synthesis of Protocol Entities' Specifications from Service Specifications in a Perti Net Model with Registers,” *Proc. of 15th Int. Conf. on Distributed Computing Systems (ICDCS-15)*, pp. 510–517, 1995.
16. H. Kahlouche and J. J. Girardot, “A Stepwise Requirement Based Approach for Synthesizing Protocol Specifications in an Interpreted Petri Net Model,” *Proc. of INFOCOM '96*, pp. 1165–1173, 1996.
17. A. Al-Dallal and K. Saleh, “Protocol Synthesis Using the Petri Net Model,” *Prof. of 9th Int. Conf. on Parallel and Distributed Computing and Systems (PDCS'97)*, 1997.
18. A. Khoumsi and K. Saleh, ”Two Formal Methods for the Synthesis of Discrete Event Systems,” *Computer Networks and ISDN Systems*, Vol. 29, No. 7, pp. 759–780, 1997.
19. M. Kapus-Koler, “Deriving Protocol Specifications from Service Specifications with Heterogeneous Timing Requirements,” *Proc. of 1991 Int. Conf. on Software Engineering for Real Time Systems*, pp. 266–270, 1991.
20. A. Khoumsi, G. v. Bochmann and R. Dssouli, “On Specifying Services and Synthesizing Protocols for Real-time Applications,” *Proc. of 14th IFIP WG6.1 Symp. on Protocol Specification, Testing and Verification (PSTV-14)*, pp. 185–200, 1994.
21. A. Khoumsi and G. v. Bochmann, “Protocol Synthesis Using Basic LOTOS and Global Variables,” *Proc. of 1995 Int. Conf. on Network Protocols (ICNP'95)*, 1995.
22. A. Nakata, T. Higashino and K. Taniguchi, “Protocol Synthesis from Timed and Structured Specifications,” *Proc. of 1995 Int. Conf. on Network Protocols (ICNP'95)*, pp. 74–81, 1995.
23. H. Yamaguchi, K. Okano, T. Higashino and K. Taniguchi, “Protocol Synthesis from Time Petri Net Based Service Specifications,” *Proc. of 1997 Int. Conf. on Parallel and Distributed Systems (ICPADS'97)*, pp. 236–243, 1997.
24. J. -C. Park and R. E. Miller, “Synthesizing Protocol Specifications from Service Specifications in Timed Extended Finite State Machines,” *Proc. of 17th Int. Conf. on Distributed Computing Systems (ICDCS-17)*, 1997.
25. K. El-Fakih, H. Yamaguchi and G.v. Bochmann, “A Method and a Genetic Algorithm for Deriving Protocols for Distributed Applications with Minimum Communication Cost,” *Proc. of the 11th IASTED Int. Conf. on Parallel and Distributed Computing and Systems (PDCS'99)*, 1999.
26. H. Yamaguchi, K. El-Fakih, G.v. Bochmann and T. Higashino, “A Petri Net Based Method for Deriving Distributed Specification with Optimal Allocation of Resources,” *Proc. of the ASIC Int. Conf. on Software Engineering Applied to Networking and Parallel/ Distributed Computing (SNPD'00)*, pp. 19–26, 2000.
27. S.S. Skiena, “The ALGORITHM Design Manual,” *TELOS - The Electronic Library of Science (A Springer-Verlag Imprint)*, 1998.
28. Kellner, M. et al. : “ISPW-6 Software Process Example,” *Proc. of the 1st Int. Conf. on the Software Process*, pp. 176-186, 1991.
29. “lp_solve,” `ftp://ftp.ics.ele.tue.nl/pub/lp_solve/`
30. G. Rothermel and M. J. Harrold, “Analyzing Regression Test Selection Techniques,” *IEEE Trans. on Software Engineering*, Vol. 22, No. 8, pp. 529–551, 1996.