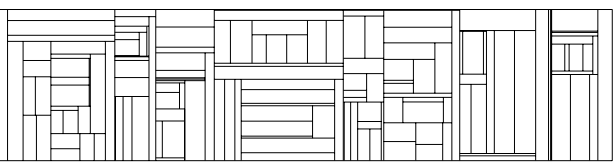
ISSN 0105-8517

Workshop on Modelling of Objects, Components, and Agents Aarhus, Denmark, August 27-28, 2001

Daniel Moldt
(Ed.)

DAIMI PB – 553
August 2001

DATALOGISK INSTITUT
AARHUS UNIVERSITET
Ny Munkegade, Bygn. 540
8000 Århus C



Preface

This booklet contains the proceedings of the workshop Modelling of Objects, Components, and Agents (MOCA'01), August 27-28, 2001. The workshop is organized by the „Coloured Petri Net" Group at the University of Aarhus, Denmark and the „Theoretical Foundations of Computer Science" Group at the University of Hamburg, Germany. The papers are also available in electronic form via the web pages: http://www.daimi.au.dk/CPnets/workshop01/

Objects, components, and agents are fundamental concepts often found in the modelling of systems. Even though they are used intensively in software engineering, the relations and potential of mutual enhancements between Petri-net modelling and the three paradigms have not been finally covered. The intention of this workshop is to bring together research and application to have a lively mutual exchange of ideas, view points, knowledge, and experience.

The submitted papers were evaluated by a programme committee consisting of:

Wil van der Aalst	(The Netherlands)
Remi Bastide	(France)
Jonathan Billington	(Australia)
Didier Buchs	(Switzerland)
Henrik Bærbak Christensen	(Denmark)
Jose-Manuel Colom	(Spain)
Jörg Desel	(Germany)
Susanna Donatelli	(Italy)
Nisse Husberg	(Finland)
Ekkart Kindler	(Germany)
Gabriela Kotsis	(Austria)
Fabrice Kordon	(France)
Charles Lakos	(Australia)
Rainer Mackenthun	(Germany)
Daniel Moldt (Chair)	(Germany)
Kjeld Hoyer Mortensen	(Denmark)
Dan Simpson	(United Kingdom)
Rüdiger Valk	(Germany)
Tomas Vojnar	(Czech Republic)

The programme committee has accepted 8 papers for presentation. They tackle the concepts of objects, components, and agents from different perspectives. Formal as well as application aspects demonstrate how wide the range can be within which Petri nets can be used and illustrate at the same time that there is a tendency to use more abstract concepts for the analysis and design of Petri-net-based models.

Daniel Moldt

Content

Towards an Adequate Framework for Specifying and Validating Runtime Evolving Complex Discrete-event Systems

Nasreddine Aoumeur Gunter Saake
Institut für Technische Informationssysteme
Otto-von-Guericke-Universität Magdeburg
Postfach 4120, D–39016 Magdeburg
E-mail: {aoumeur|Saake}@iti.cs.uni-magdeburg.de
Tel.: ++49-391-67-18659 Fax: ++49-391-67-12020

Abstract

Although formal specification / validation is nowadays definitely accepted as a necessary step in developing *reliable* complex discrete-event systems, most of existing formalisms are facing challenging problems due to the multi-dimensional requirements to be met by such systems. As a result of this unsatisfactory state of affair, different system's facets are often separately approached by different formalisms which avoid any coherent description of the whole system. Besides that, most of formal proposals are lacking conceptual means for dynamically updating the (initial) system's specification as unavoidable consequences of technological change, user's requirement modification, etc.

As a contribution towards a satisfactory framework, we present in this paper a multi-paradigm formalism that we argue allows coping in a coherent way with most requirements in complex (discrete-event) systems. This formalism, referred to as evolving CO-NETS, soundly integrates: (1) Concepts from object-orientation and modularity principles for coping with the complex structure and behaviour of such systems; (2) Timed high level Petri-nets for intrinsically mastering the concurrent and time depending nature of such systems; (3) A true-concurrency rewriting logic-based semantics for capturing in a simple way runtime behaviour modification in this proposed object-based Petri nets, for deriving rapid-prototypes using rewriting techniques, and last but not least for controlling the behaviour using strategies—possible due to the reflective nature of this logic. All key features of this framework are illustrated using a non-trivial version of the European train control systems (ETCS).

1 Introduction

Rigorous specification and validation, before any efficient and user-friendly implementation, represents nowadays more than ever the most *crucial* phase in any complex software development. Moreover, such a precise description of future software facets with validation and verification of their important properties becomes *unavoidable* for software dedicated to critical systems, where any misconception results in disastrous consequences.

Unfortunately, despite the existing of several formal approaches we are far from any widely accepted formalism which intrinsically copes with all dimensions characterizing present-day critical systems. Indeed with most of existing approaches, system's facets are often separately approached by different formalisms avoiding any coherent description and analysis of the whole system.

As a contribution to this vivid and complex area of research, we present in this paper a new proposal that we argue allows to cope coherently with the following requirements in complex critical systems: complex data and knowledge, concurrent and and distributed behaviour, real-time constraints, and their evolution in a non-previous way.

The model we are proposing is referred to as ECO-NETS (an acronym for evolving concurrent object-oriented Petri nets). More precisely, with respect to the above multi-dimensional requirements

the main features of this approach may be highlighted as follows.

Object orientation with modularity: For coping with the first dimension, we take profit of all object-oriented abstraction mechanisms (i.e. object, classification, different forms of inheritance, object-composition and object interaction). Moreover, to allow an incremental description for such systems, we regard each class rather as a module with an explicit interface including observed attributes and imported / exported messages. This notion of module is naturally extended to the notion of a component as hierarchy of modules using different forms of inheritance and object composition.

Timed high-level Petri nets: For adequately dealing with timing constraints with a full exhibition of intra- and inter-object concurrency with different forms of communication, we integrate all mentioned object-oriented structuring mechanisms into an appropriate variant of timing algebraic Petri nets. This integration has been firstly introduced in [2].

Meta-reasoning on object Petri nets: For coping the dynamic evolution in complex discrete-event systems, we propose a natural extension of this object Petri nets variant. The main ideas firstly forwarded in [1] consists in: (1) introducing meta-places those tokens are a complete transition's behaviour; associate three (meta-)transitions for updating, deleting or creating any behaviour as tokens; (3) relate this meta-level with the object level through appropriate read-arcs.

Rewriting logic-based semantics: For rapid-prototyping purposes and for a clean true concurrency semantics, we interpret the behaviour of different transitions in rewriting logic [12]. Also, for capturing the runtime behaviour we propose a two-step valuated inference rule in this logic. The other crucial advantages of this logic is its intrinsic reflective nature [6]. We take advantages of this property for composing different transitions behaviour in a runtime way, leading to different strategies which may manifest such systems.

The proposed approach, namely ECo-nets is incrementally introduced using a non trivial version of the European train control systems (shortly ETCS). This case study has been established as one of two applications in a (DFG) project[1]. This complex case study manifests all the mentioned crucial features of critical systems. Indeed, first, it is composed of several synchronously and/or asynchronously communicating components including: trains, gates, sensors, control-light and software controllers. Secondly, each component includes complex multi-layered data and concurrent behaviour, and shows only part of them to other components. Thirdly, all components work under very precise and critical timing constraints. Fourthly, the technology used in each component is rapidly changing towards a more and more sophisticated ones.

The rest of this paper is organized as follows. In the second section we informally introduce the ETCS case study we deal with in this paper. The third section presents how different components are formally conceived using the Co-nets approach. In the next section we concentrate on the runtime behaviour modification in Co-nets using this running application, leading to the introduction ECo-nets. In the fifth section, using the reflective level of rewriting logic, we show how strategies over rewrite rules gouverning ECo-nets behaviour may be expressed. The last section summarizes the achieved work, and outlines our future work. We point out however that due to space limitation our presentation is mostly intuitive; corresponding rigorous definitions of all concepts may be found in [3].

[1] "Integration of Software Specifications for Engineering Applications" supported by the German Research Community (DFG).

2 The European Train Control System (ETCS)

2.1 ETCS problem : technical presentation

In its technical abstract version [15] as a generalization of the railroad crossing systems [8], the ETCS problem can be expressed as follows. A railroad crossing with several train tracks and common gates can be represented as depicted in Figure 2.1. Sensors along every track detect oncoming and departing trains. Based on signals from these sensors, an automatic *controller* signals the gate to open or close, and also manages light-control in consequence. After receiving a signal form a sensor, the train enters in what is called the region of interest (shortly, R); afterwards depending on the state of the gate the train has to stop or to cross the road (I).

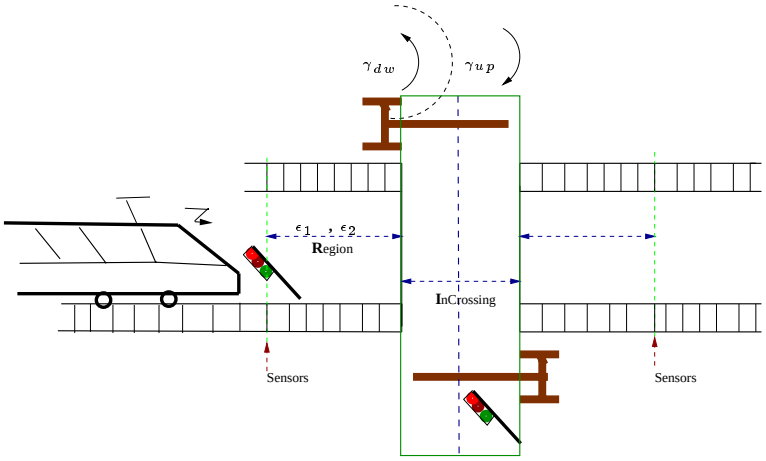


Figure 1: An abstract schema of a railroad crossing.

The technical timing requirement may be expressed as follows.

- ϵ_1 (resp. ϵ_2) represents the lower (resp. the higher) timestamp for the train to reach the road after being detected by a sensor (or to be detected by a sensor after leaving the road). These two limits depend among other on the current speed of the train.
- $I \subseteq R$; $g(t) \in [0, 90]$; with $g(t) = 0$ corresponding to gate down.
- $\{\lambda_i\}$ of *occupancy intervals* : one or more trains are in I. ith occupancy is represented as $\lambda_i = [\tau_i, \nu_i]$. τ_i is the time of the ith entry of a train into the crossing when no other train is in the crossing. ν_i is the first time since τ_i that no train is in the crossing.
- Given two constants $\xi_1 > 0$ and $\xi_2 > 0$ (with $\gamma_{dw} + \epsilon_1 < \xi_1 < \gamma_{dw} + \epsilon_2$ and $\gamma_{dw} + \epsilon_1 < \xi_2 < \gamma_{up} + \epsilon_2$), the problem is to develop a system to operate the crossing gate that satisfies the following two properties:

  Safety Property: $t \in \cup_i \lambda_i \Rightarrow g(t) = 0$ (the gate is down during all occupancy intervals.)

  Liveness Property: $t \notin \cup_i [\tau_i - \xi_1, \nu_i + \xi_2] \Rightarrow g(t) = 90$. (the gate is up when no train is in the crossing)

2.2 ETCS : an informal OO description

In this version of ETCS we consider five components—as classes with encapsulated behaviour and explicit interfaces. Four components belong to the environment, namely *Train* describing trains traveling on controlled tracks, *Gate* as operating road-crossing gates, *Sensors* for automatically detecting trains approaching the gates, and *Light-Control* for managing the lights at each gate. The *Controller*

component corresponds to the controlling software system to be installed for operating the gates. Using OO notations adopted from [16] and slightly adapted for emphasizing the distinction between any component's local and observed features and the inter-component interaction, we depicted in Figure 2 a diagrammatic OO description of this ETCS system.

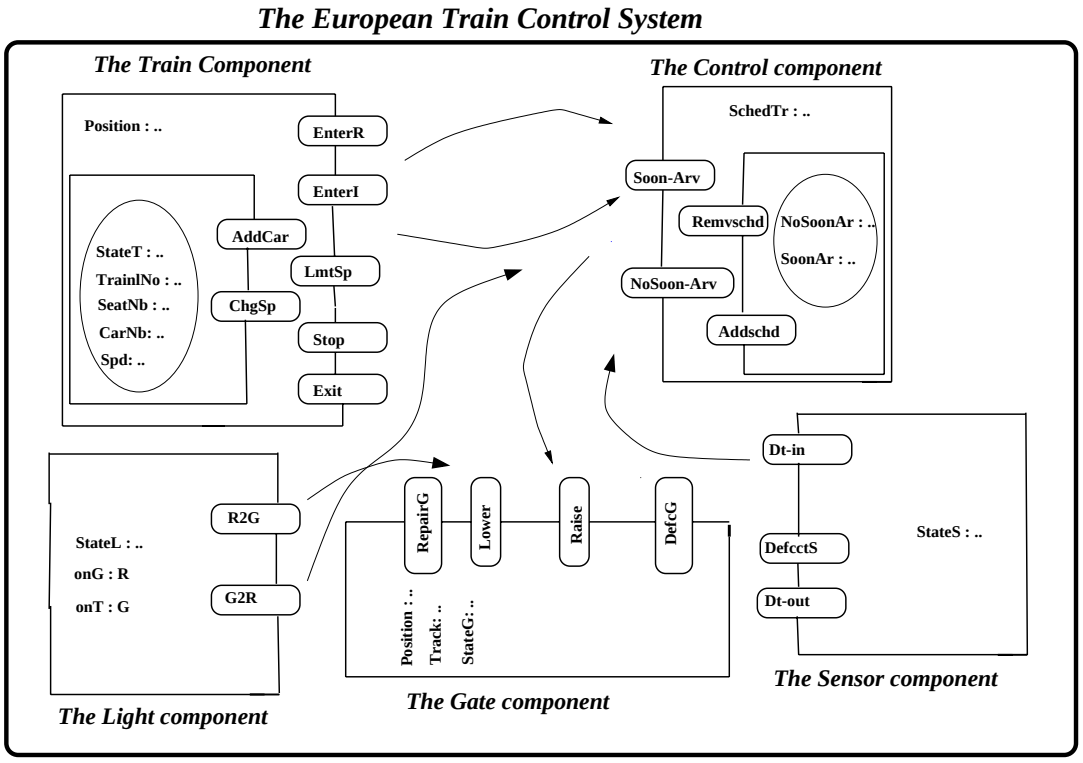


Figure 2: Informal description of different ETCS components

In some details, attributes and method-invocations or messages of each component are as follows:

Train component: Besides the self-explained (local) attributes —$TrainNo$, $SeatNb$, $CarNb$—, there is the position *pos* which is an observed one. This attribute may have three values: E(lsewhere) ("not in the section of interest"), R ("in the region of interest"), or I ("in the railroad crossing"). Also, the current speed is represented by *Spd*. Finally, by $StateT$ we refer to the state of train which may be *on* (i.e. traveling) or *off* expressing that the train is stopped due to gate failure for instance. The operations acting on this (train-) state are the following. *EnterR* allows for changing the attribute position from E to R, and at the same time informs the *Controller* component of this arrival. *EnterI* (resp. *Exit*) corresponds to the entry (resp. leaving of) in the railroad. Besides that, we have included an operation *AddCar* that allows for adding new car (with a corresponding number of sites)[2]. Also, when the train received (from other components) a *stop* message its state becomes *off*.

Gate component: This component is mainly characterized by the position of the gate that may be *Up, Down* (shortly, *Dw*), and by the state of each track (i.e. occupied or free, shortly *Oc, fr*). That is, in this version we allow a gate with several tracks. The associated messages on this state are *Lower, Raise* which have to be sent by the *Controller* component, and also a systematic appearance of a *defect* message in case of a breaking-down of the gate. In such case this message is first communicated to the controller component and then broadcasted to all trains, scheduled for this gate, to be stopped until the gate is being repaired; thus we have also another message denoted *repair*.

[2] This operation is irrelevant for the system, but it allows us to make explicit the intra-object concurrency exhibition.

Controller component: It uses as observed attributes: *SchedTr* a set of already scheduled trains (with corresponding crossing times and associated gate and track) allowed to cross the gate, an (boolean) attribute *SoonArrival* indicating that a train will soon arrive (in less than *Gamma* units of time), and another boolean attribute *NoSoonArrival* indicating that no train is scheduled for a sufficient time (i.e. more than *Gamma*) in which case the gate should be *Up*. Two messages are to be considered in this component. First the scheduling of a given train (i.e. its addition to the *SchedTr* list). Each schedule is a tuple composed of: train identity, corresponding gate, associated track in such gate, and the time to pass this gate. Second the removal of a scheduled train after passing a corresponding gate. After passing a sensor, we require that the speed of the train should be less than some constant Sp_1; when a sensor is displaying a defect state such speed should be less than Sp_2 (with $Sp_2 < Sp_1$)
Sensor component: This component is mainly characterized by the state of the sensor that may be *Ok* or defect (shortly, *Dfc*). The associated messages on this state are *detect* and *defect*; where *detect* returns the time and the corresponding train, whereas *defect* indicates that the corresponding sensor is not working.
Light-control component : This component is mainly characterized by the state of the light system (i.e. as for sensor *Ok, Dft*), the color of light in direction of the gate as well as the one in direction of the tracks (shortly, *onG, onT*. The methods that may exhibit this component are the change between different colors (i.e. *Red-Yellow-Green*).

3 CO-NETS for components specification and validation

The purpose of this section is to review and apply, in an incremental way,[3] to this ETCS case study the main CO-NETS concepts for deriving a formal component-based specification which can in addition be directly validated. First, the notion of component signature is presented and applied to derive a formal specification of different ETCS component signatures. Second, we highlight how the (concurrent) behaviour of different components is then derived. Third, we present how rewrite rules gouverning such behaviour are obtained. Fourth, the general pattern for interacting different components through their observed part is sketched and applied to the ETCS problem. We note however that due to space limitation inheritance is not addressed, and only some components are thoroughly specified.

3.1 CO-NETS component signature

The component signature defines the structure of object states and operations to be accepted by such states. The OO signature that we propose can be informally[] described as follows:

- Object states are (algebraic) terms as tuples of the form: $\langle Id | atr_1 : val_1, .., atr\_k : val_k, at\_bs_1 : val'_1, .., at\_bs_{k'} : val'_s \rangle$; where Id is an observed object identity; $atr_1, .., atr_k$ are local attribute identifiers having as current values respectively $val_1, .., val_k$. Attributes observed by other components are identified by $at\_bs_1, ..., at\_bs_s$ with $val'_1, .., val'_k$ as respective current values.

- This object state is allowed to be split (resp. recombined) at a need. This 'splitting / recombination' is regarded as an axiom, which may take the form

$$\langle Id | attrs_1, attrs_2 \rangle = \langle Id | attrs_1 \rangle \oplus \langle Id | attrs_2 \rangle$$

  with $attr_i$ as an abbreviation of $atr_{i1} : val_{i1}, ..., atr_{ik} : val_{ik}$. The multiset operator $\oplus$ will be explained later.

- Messages are just operations but with at least one argument of object state sort, that is, they should be sent or received by objects. We also make a clear distinction between local messages and external ones, as imported or exported messages. Local messages trigger object states change in a given component, whereas external ones allow for interacting different components.

[3] Which can be seen as inherent methodology for constructing complex systems using CO-NETS approach.

We note that for describing component signature, we adopt as syntactical notations an OBJ [7] algebraic language notation-like.

3.1.1 Signatures of different ETCS components

For the ETCS problem five CO-NETS component signatures have to be specified. But first we have to specify the common (algebraic) data level for different object values and message parameters. We note that just the train component signature is described below; the other signatures are sketched in the appendix.

In the data level specified below, there are particularly the OId sort which stands for the object identity sort, the $Time$ that we assume is specified elsewhere, and the different states of the gate (i.e. GUp for going up, Up, etc). Finally, we have also the scheduled set of trains as a list of tuples of tuples of the form $[OId\_of\_train, [OId\_of\_gate1, tracks-number1, crossing\_time1]|[OId\_of\_gate2, tracks-number2, crossing\_time12]|...]$.... That is for each train we group together all road-crossings (information) it is going to pass through. This facilitate, for instance, any removal as well as rescheduling due to gate failure.

```
obj ETCS_Sturcture is
  protecting OId Time Bool Nat Real .
  sorts Gate   PositionT   PositionG .
  subsorts OId-Gate  OId-Train < OId .
  subsort Track-Nb < Nat .
  sorts SchedSet Elt Color.
  subsorts T-roadElt < T-roadList.
  subsort nil < SchedSet .
  subsort Sched-Elt < SchedSet .
  op :  GUp, Up, GDw, Dw :  → PositionG .
  op Gr, Red :  → Color .
  op :  Gamma :  → Time .
  op :  [_,_,_]:  OId-Gate Track-Nb Time → T-roadElt .
  op :  _|_ :  T-roadElt  T-roadList → T-roadList .  [assoc.  comm.]
  op :  [_,_] :  OId-Train  T-roadList → Sched-Elt .
  op :  _._ :  SchedSet SchedSet
   → SchedSet [assoc.  comm.]  .
  op :  _in_:  Elt SchedSet → Bool .
  var S : SchedSet .
  vars E, E' : Elt .
  vars Now, t :  Time .
  eq E in nil = false .
  eq E in E'.S =
   if E == E' then true else E in S fi.
endo.
```

Train component signature: From the above OO informal description of the ETCS, the formal train component signature may be directly derived as follows.

```
obj Train is
  protecting ETCS_Structure .
  subsort Id.Train < OId .
  subsort Local_Train  External_Train < Train .
  subsort EnterI  AddCar < Local_Train_Mes .
  subsort EnterR  Exit < Exported_Train_Mes .
  (* local attributes *)
   op ⟨_|Spd : _, NbSeat : _, NbCar : _⟩ :
    Id.Train  Real  NAT  NAT → Local_Train.
  (* observed attributes *)
   op ⟨_|Pos : _, StateT : _⟩ :   Id.Train  POSITION → External_Train .
  (* local messages *)
   op AddCar:  Id.Train  Nat → Local_Train_Mes .
   op ChgSp:  Id.Train Real → Local_Train_Mes .  (* for speed change *)
  (* Exported messages *)
```

```
  op EnterR: Id.Train  OId → Exported_Train_Mes
  op Exit:  Id.Train  OId → Exported_Train_Mes
  op EnterI: Id.Train → Exported_Train_Mes .
  op LmtSp:  Id.Train  Real → Exported_Train_Mes (* for speed limitation *).
  vars T : Id.Train .(* to be used in the net *).
  vars P, P1 :  REAL.
endo.
```

3.2 CO-NETS component specification

On the basis of a given component signature, we define the notion of component specification as a CO-NET in the following straightforward way.

- CO-NETS places are precisely defined by associating with each message generator one (message) place, that is, such messages places contain associated message instances sent or received by objects but not yet performed (i.e. by firing their corresponding transitions). Also, with each object sort a (object) place is associated, that is, an object place contains current object states with respective values. We note that places corresponding to external messages will be drawn with bold circles.

- CO-NETS transitions reflect the effect of messages on object states to which they are addressed. Conditions may be associated to them restricting their application. Moreover, we distinguish between local and external transitions. Local transitions reflect object states change in a given component, whereas external ones capture the interaction between different components. For both we propose in next section appropriate general patterns.

- We deal with the real-time constraints in our approach by attaching timestamps or time internals to transitions and to time-dependent messages. An appropriate semantics, such time-dependent transitions will be interpreted in timed rewriting logic [10].

3.2.1 ETCS component specification in CO-NETS

As for the ETCS component signatures, we comment here with some details on the Train component whereas the other component specifications are sketched in the appendix.
The Train CO-NET component: This net as depicted in Figure 3 is composed of an object place denoted by `TRAIN` containing current (object) states of different trains, three places corresponding to different local messages, and four places associated with observed messages (depicted in the left hand side with bold lines). The effect of each message is captured by an appropriate transition. For instance, the effect of the message `EnterR(T)` is captured by the transition `ENTER-R` which takes as input the position of the train which should be `E`(lsewhere). After the firing of this transition, there is a sending of `EnterI(T)` message and change of the position to `R`(in the region of interest) (i.e. $\langle T|Pos : R\rangle$)[4]. It is also worthwhile noting that transitions labeled by `EnterI`, `Enter-R` and `Exit` are time-dependent transitions as required in the informal (technical) description.

3.3 CO-NETS Component semantics

After highlighting how CO-NETS components are constructed, we focus herein on the behavioural aspects, that is, how object states' components have to change in a *coherent* and true concurrent way. By coherent we specifically understand the respect of object uniqueness and the encapsulation property. For this aim, we first propose a general pattern that has to be respected in constructing transitions in such components (i.e. local transitions). Second, a rewrite rule is derived for each transition on the basis of this general intra-component transition pattern.

[4] Remark that due to the splitting axiom only the position which is the just concerned attribute in this effect is selected.

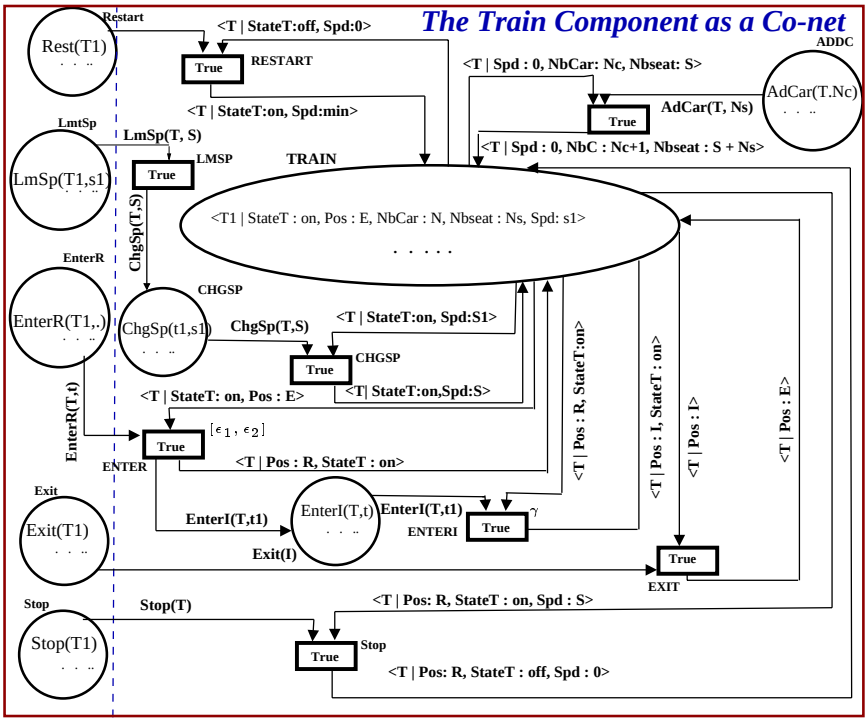


Figure 3: The Train Component as a Co-net.

This general transition pattern is depicted in Figure 4, and it can be intuitively explained as follows: The contact of just the relevant parts of some object states in a given component Cp, —namely $\langle I_1|attrs_1\rangle$[5] ;..; $\langle I_k|attrs_k\rangle$— with some messages $ms_{i_1}, .., ms_{i_p}$ declared in this component results in the following effects: (1) the messages $ms_{i1}, .., ms_{ip}$ disappear; (2) some (parts of) object states participating in the communication change, namely $\langle I_{s_1}|attrs_{s_1}\rangle, ..., \langle I_{s_t}|attrs_{s_t}\rangle$; (3) some objects may be explicitly deleted by sending them delete messages; and (4) new messages may be sent to objects of Cp, namely $ms'_{h_1}, .., ms'_{h_r}$.

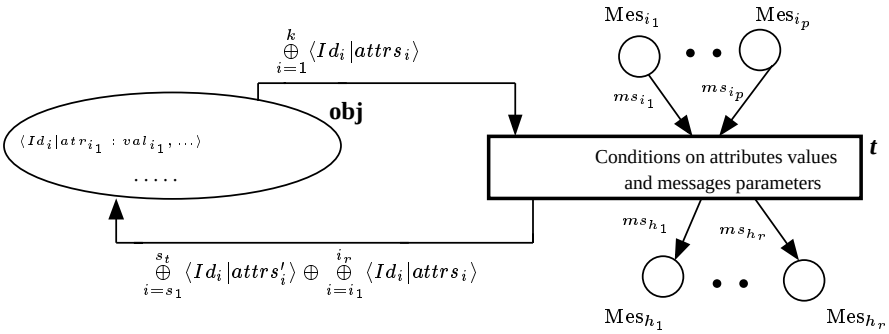


Figure 4: A general intra-component or local transition pattern

We note in this pattern that markings in different places is as usual regarded as a multiset of tokens. Such (marking) multiset is constructed using a union operator we denote by $\oplus$.

3.3.1 Rewrite theory for Co-nets semantics

As we mentioned in order assign to Co-nets behaviour a true-concurrency semantics with full exhibition of intra- and inter-object concurrency instead of an interleaving or even a step-based semantics like

[5] With $attrs_i$ as a simplified notation of $atr_{i1} : val_{i1}, .., atr_{ik} : val_{ik}$.

for colored Petri nets [9], each transition is to be gouverned by a corresponding rewrite rule interpreted in rewrite logic [12]. The main ideas (adapted from the algebraic net in [4]) consist in: (1) associate with each marking mt its corresponding place p as a pair (p, mt); (2) to capture the current state of a CO-NETS as multiset over different pairs (p_i, mt_i) we introduce a new multiset generated by a union operator we denote by $\otimes$, that is, a CO-NETS state is hence described as $(p_1, mt_1) \otimes (p_1, mt_2) \otimes ...$; (3) in order to exhibit a maximal of concurrency we require the distributivity of $\otimes$ over $\oplus$, that is, if mt_1 and mt_2 are two marking multisets then we always have: $(p, mt_1 \oplus mt_2) = (p, mt_1) \otimes (p, mt_2)$. Finally, intra-object concurrency is to be ensured by the splitting / recombining axiom.

On the basis of these more or less intuitive ideas, the general rewrite rule gouverning the general transition pattern depicted in Figure 4 take the following form:

t: $(obj, \overset{k}{\underset{i=1}{\oplus}} \langle Id_i | attrs_i \rangle) \overset{p}{\underset{k=1}{\otimes}} (Mes_{ik}, ms_{ik}) \Rightarrow (obj, \overset{t}{\underset{k=1}{\oplus}} \langle Id_{s_k} | attrs'_{s_k} \rangle \oplus \overset{r}{\underset{k=1}{\oplus}} \langle Id_{i_k} | attrs_{i_k} \rangle)$
$\overset{r}{\underset{k=1}{\otimes}} (Mes_{h_k}, ms_{h_k})$ `if` *Condition.*

3.3.2 Application for deriving Train transitions rewrite rules

By applying this general form of rule, it is not difficult to generate the rules governing ETCS components' behaviour. For instance, the rewrite rules associated with the train component are as follows:

ADDC: $(Train, \langle T | Spd : 0, NbCar : Nc, Nbseat : S \rangle) \otimes (AddC, AddCar(T, Ns)$
$\Rightarrow (Train, \langle T | Spd : 0, NbCar : Nc + 1, Nbseat : S + Ns \rangle)$

ENTER: $(Train, \langle T | Pos : E, StateT : on \rangle) \otimes$
$(EnterR, EnterR(T, C)) \overset{[\epsilon_1, \epsilon_2]}{\Rightarrow} (Train, \langle T | Pos : R, StateT : on \rangle)$
$\otimes (EnterI, EnterI(T))$

ENTERI: $(Train, \langle T | Pos : R, StateT : on \rangle) \otimes (EnterI, EnterI(T))$
$\overset{\gamma}{\Rightarrow} (Train, \langle T | Pos : I, StateT : on \rangle) \otimes (Exit, Exit(T, C))$

EXIT: $(Train, \langle T | Pos : I, StateT : on \rangle) \otimes (Exit, Exit(T, C))$
$\Rightarrow (Train, \langle T | Pos : E, StateT : on \rangle) \otimes (Exit, Exit(T, C))$

STOP: $(Train, \langle T | Pos : R, StateT : on, Spd : S \rangle) \otimes (Stop, Stop(T))$
$\Rightarrow (Train, \langle T | Pos : R, StateT : off, Spd : 0 \rangle)$

LMSP: $(LmtSp, LmtSp(T, S) \Rightarrow (CHGSP, ChgSp(T, S)$

CHGSP: $(CHGSP, ChgSp(T, S) \otimes (Train, \langle T | StateT : on, Spd : S_1 \rangle)$
$\Rightarrow (Train, \langle T | StateT : on, Spd : S \rangle)$

It is worth observing that a timestamp (γ) and time-interval ($[\epsilon_1, \epsilon_2]$) label the two time-dependent transitions, namely $ENTER$ and $ENTERI$. Such timed-rewrite rules are interpreted in Timed rewrite logic [17] as a straightforward extension of rewrite logic [12]. Using the four inference rules of this logic and particularly the concurrent replacement ones (with the state splitting / merging axiom and the distributivity of $\otimes$ over $\oplus$) any reachable state can be computed in true concurrent way starting from an initial marking for different places. Moreover, by simultaneously accompanying such computation with graphical simulation most of misconception, errors and missing can be detected and then corrected.

3.4 CO-NETS components interaction

Taking into account that object state change in each component is ensured by the already described intra-component pattern, the inter-component interaction may be made explicit as follows: The contact of some *external* parts of some object states, namely $\langle I_1 | attrs\_ob_1 \rangle$, ...,$\langle I_k | attrs\_ob_k \rangle$, which may belong to different components namely $Cp_1, ..., Cp_m$ with some *external* messages $ms_{i1}, .., ms_{ip}$ results in the following: (1) the messages $ms_{i1}, .., ms_{ip}$ disappear; (2) some external parts of object states participating in the communication change;(3) new external messages (that may involve deletion/creation ones) are sent to objects in different components, namely $ms'_{h1}, .., ms'_{hr}$. This inter-component interaction or external transition general pattern is depicted in Figure 5.
The rewrite rule associated with this transition general form presents no particular treatment, and is deduced exactly in the same way as for the intra-component pattern.

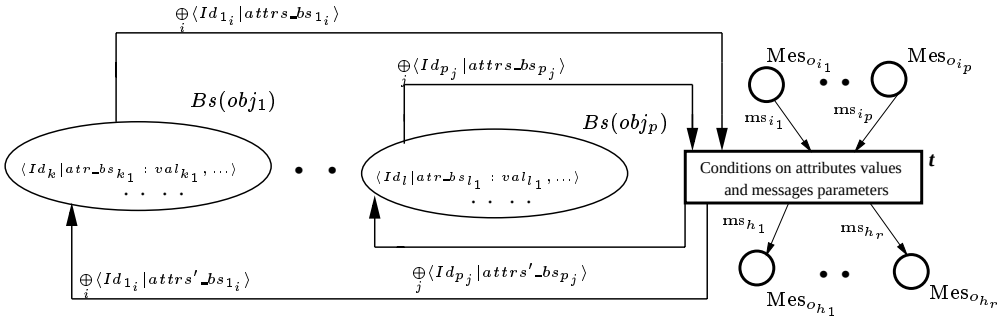


Figure 5: The Inter-component interaction pattern

3.4.1 Interacting different ETCS components

To achieve the desired behaviour in the ETCS problem, the five components have to interact, with respect to this inter-component pattern, using their imported / exported messages and observed attributes. This interaction is depicted in Figure 6. In this interaction, for instance, the transition `Dt-I` takes as inputs: (1)the detection by sensors of a given train `T` approaching a gate `G` on a track `K`(i.e as input message `Dt-in(T,G,K,t)`); (2) and the effective scheduling of such train at this time `t` (using the exported message (`Sn(T,G,K,t)`). The resulting output messages of this transition are: (1) a sending to the train component of the message `EnterR(T,G,K,now)` (i.e. entering into the region of interest 'R'); (2) a sending of a message for limiting train speed to $\texttt{Sp}_1$; and (3) a sending of message to the light component for changing its light color from green to red. The remaining transitions are also constructed following the same reasonning.
The rewriting rules governing this interaction can be captured in a similar way, as done for the internal behaviour, from the effect of each transition.

4 Runtime Evolution in Co-nets

The purpose of this subsection is, first, to review the main ideas and corresponding constructions for handling runtime modification that we first proposed in [1]. Then, we propose a more adequate inference rule for propagating a given behaviour from the meta-level to the object level. Finally, we show how such meta-level allows for an ETCS evolving specification.

4.1 Meta-places and non-instantiated transitions constructions

For handling runtime modification of Co-nets component specifications, the constructions we proposed in [1] may be summarized as follows:

1. In order to free some Co-nets transitions[6] from their rigidity, we propose to *replace* each of their three components— namely input tokens inscribing their input arcs, output tokens inscribing their output arcs and their conditions— by appropriate variables those sorts are exactly the sorts of associated input or output places. We refer to such transitions with only variables as inscriptions as *non-instantiated* transitions. Their general form is sketched in the lower right hand-side of Figure 7. In this general pattern for non-instantiated transitions, all (arc-) inscriptions, namely $IC_{obj}, IC_{i_1}, .., IC_{i_p}$ for input arcs, $CT_{obj}, CT_{h_1}, .., CT_{h_r}$ for output arcs and TC for the condition are to be considered as appropriate variables.

2. Second, for each becoming non-instantiated transition we gather its initial (i.e. before their replacement by variables) condition and arc inscriptions with their respective input /output places into a a single *tuple* of the form: ⟨`transition_id: version | (input-)multiset,`

[6] In the same spirit as in [14], we assume that some behaviour, i.e. transitions, is fixed forever reflecting minimal properties of the specified application.

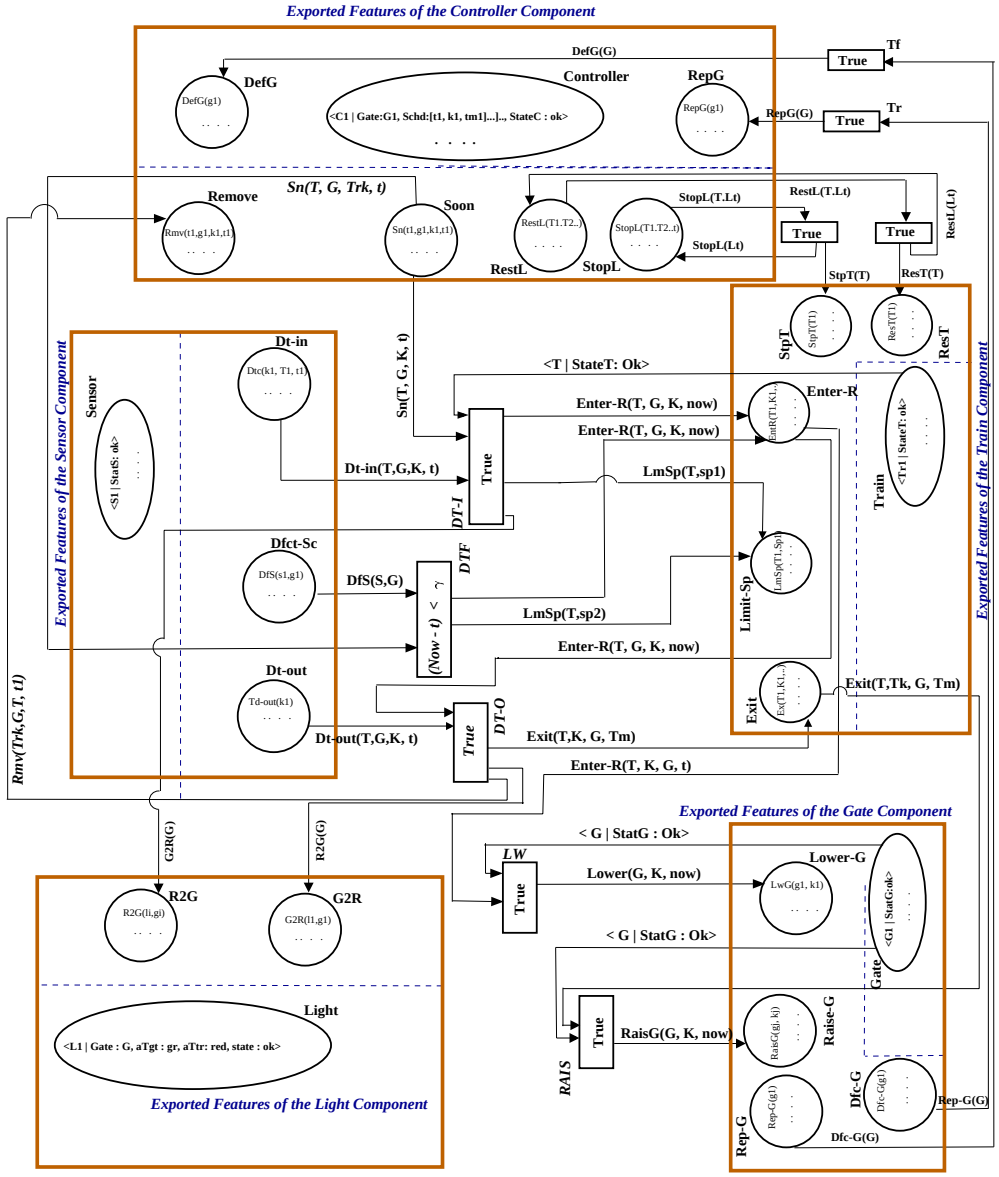


Figure 6: The Train-Gate-Controller-Sensor-Light Interacting using observed features

`(output-)multiset, condition` $\rangle$. In this tuple `transition_id` refers to the label of the transition, whereas `version` is just a natural number indicating that this tuple can be regarded as a particular version of a behaviour associated with this transition; this will be more clear in the following. In particular, the precise form of such a tuple with respect to the transition (general-pattern) T in Figure 4 takes the following form; where the index i represents a particular *version* of such transition.

$$\langle T : i \mid (obj, IC_{obj}) \overset{i_p}{\underset{k=i_1}{\otimes}} (Mes_k, IC_k), (obj, CT_{obj}) \overset{h_q}{\underset{k=h_1}{\otimes}} (Mes_k, CT_k), Condition \rangle$$

3. Third, we consider such 'behaviour' tuples as tokens w.r.t. a new place, namely `meta-place` in Figure 7. This place constitutes the first element of the proposed meta-level. On the basis of this behaviour as tokens, it becomes quite possible to delete some of them, modify some of them or introduce new ones: This corresponds respectively to the transitions `DEL`, `MODIF` and `ADD` in Figure 7 and their corresponding message places `Del-Bh`, `Chg-Bh` and `Add-bh`. We note that the transition `ADD` is in fact composed of two transitions: `ADD1` and `ADD2` corresponding to

the cases of adding a new version to an existing transition behaviour or a (first version for) new transition[7].

4. Finally, using an appropriate *read-arc* we relate the two levels, i.e. the meta-place in the meta-level with each non-instantiated transition in the object level.

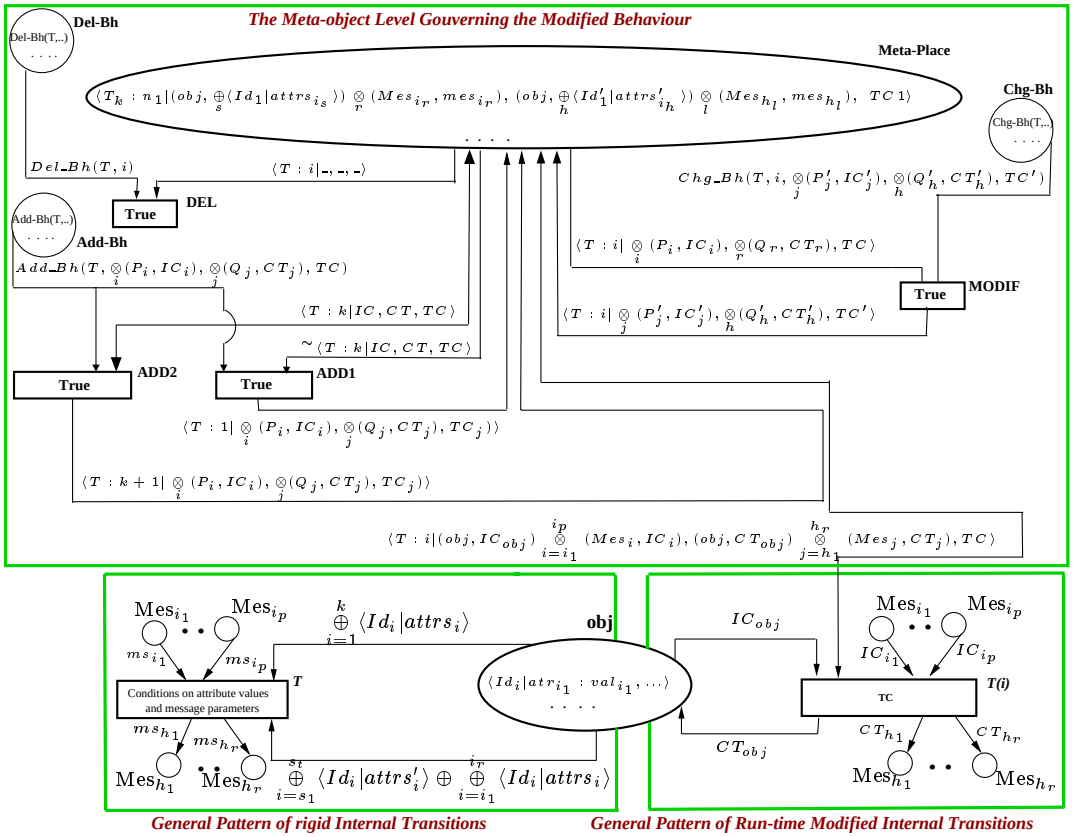


Figure 7: The general pattern for handling dynamic behaviour in CO-NETS

Remark: To deal with runtime behaviour modification also for time-dependent transition it suffices just to add another component to the tuple construction in the above conceptualization. More precisely, under the assumption that the transition T (general-pattern) in Figure 7 takes tm as unit of times or time interval then, the corresponding behaviour as tuple is:

$$\langle t : i \mid (obj, IC_{obj}) \overset{i_p}{\underset{k=i_1}{\otimes}} (Mes_k, IC_k), (obj, CT_{obj}) \overset{j_q}{\underset{k=j_1}{\otimes}} (Mes_k, CT_k), Condition, Tm \rangle$$

.

4.1.1 Semantical part : the meta-inference rule

For theoretical underpinning of these constructions, we propose with respect to the same CO-NETS rewrite logic semantic-based an adequate inference rule that can be regarded as a more flexible formulation of the one proposed in [1]. The main ideas under this reformulation as described below are the following. For each non-instantiated transition we generate a rewrite rule in the same way as we done for usual CO-NETS transitions except that we introduce a new binary operator denoted $\|_r$ separating the read-arc inscription from other pairs of place-tokens. This operator is necessary because we should express the fact that tokens selected using a read-arc are *not* from the object level (i.e. the CO-NETS

[7] The operator $\sim$ capture the notion of inhibitor arc, in the sense that the inscription should not be in the current marking.

state) but from the meta-level state. With respect to the general-form of non-instantiated transitions in Figure 7 this rule labeled by t^{nins} is derived as follows.

$$t^{nins} : (P_{meta}, \langle t : i | (obj, IC_{obj}) \overset{i_p}{\underset{i=i_1}{\otimes}} (Mes_i, IC_i), (obj, CT_{obj}) \overset{j_q}{\underset{j=j_1}{\otimes}} (Mes_j, CT_j), TC \rangle \parallel_r (obj, IC_{obj}) \overset{i_p}{\underset{i=i_1}{\otimes}} (Mes_i, IC_i)$$
$$\Rightarrow (obj, CT_{obj}) \overset{j_q}{\underset{j=j_1}{\otimes}} (Mes_j, CT_j) \quad \texttt{if} \quad TC$$

However, due to the its inference between the two-level (i.e. the meta-level through the read-arc and the object level) , this rewrite rule cannot be directly applied, and we therefore have to generate from it a usual rewrite rule or an instantiated rule we denote by $t^{ins}(i)$. This process consists in selecting through different substitutions (σ_i) a particular behaviour from the meta-place, and it can be captured through the following inference rule where $M(P_{meta})$ represents the current marking of the place `meta-place`, while the We also mention that the notation $|[T_{p_i}]|_{\oplus}$ represents a class of (multiset) term (over the associativity, commutativity of $\oplus$) those sort is exactly the one of the place p_i.

For each (meta-)rewrite rule :

$$t^{nins} : (P_{meta}, \langle t : i | (obj, IC_{obj}) \overset{i_p}{\underset{i=i_1}{\otimes}} (Mes_i, IC_i), (obj, CT_{obj}) \overset{j_q}{\underset{j=j_1}{\otimes}} (Mes_j, CT_j), TC \rangle \parallel_r (obj, IC_{obj}) \overset{i_p}{\underset{i=i_1}{\otimes}} (Mes_i, IC_i)$$
$$\Rightarrow (obj, CT_{obj}) \overset{j_q}{\underset{j=j_1}{\otimes}} (Mes_j, CT_j) \quad \texttt{if} \quad TC$$

We have:

$$\exists \sigma_o, \sigma_{o'} \in [T_{obj}]_{\oplus}, \exists \sigma_i \in [T_{Mes_i}]_{\oplus}, .., \exists \sigma_j \in [T_{Mes_j)}]_{\oplus}, \exists \sigma \in [T_{bool}]$$

$$\frac{\langle t : k \ | (obj, \sigma_o(IC_{obj})) \overset{i_p}{\underset{i=i_1}{\otimes}} (Mes_i, \sigma_i(IC_i)), (obj, \sigma_{o'}(CT_{obj})) \overset{h_q}{\underset{j=h_1}{\otimes}} (Mes_j, \sigma_j(CT_k)), \sigma(TC) \rangle}{t^{ins} : (obj, \sigma_o(IC_{obj})) \overset{i_p}{\underset{i=i_1}{\otimes}} (Mes_i, \sigma_i(IC_i)) \Rightarrow (obj, \sigma_{o'}(CT_{obj})) \overset{h_q}{\underset{j=h_1}{\otimes}} (Mes_j, \sigma_j(CT_k)) \quad \texttt{if} \quad \sigma(TC)}$$

A time-dependent adaptation of this inference rule can also be straightforwardly obtained by adding the time component.

4.1.2 Application to the ETCS case study

As we pointed out the European train control system in general are intrinsically depending to technological advances and travelers' wishes, and therefore their specification should not be fixed at once; rather it should be flexible and adaptable in consequence. Moreover, due to the vitality of the system it is more necessary to proceed such change while the system remains still working. As an application to the conceptualization we are proposing for dealing with such runtime modification, we comment in the following on how generating a flexible train component with a significant part of its behaviour becomes dynamically modifiable. Indeed, in our first CO-NETS specification of this component depicted in Figure 3, only one fixed way have been proposed for each elementary behaviour (reflected by transitions effect). This first description may rapidly become unsuitable with introduction of sophisticated trains or new regulation. We limit ourselves here to two particular cases. Firstly, it is more reasonable to regard the time-interval $[\epsilon_1, \epsilon_2]$ needed for entering the region of interest, through the transition `ENTER`, as a function of the speed, let say for instance $[(S-10) * Cst, S * Cst]$; where S is the current speed and Cst is natural constant reflecting the width of this region. Secondly, it is more logical to impose a maximal speed in transitions changing speeds (i.e. transition `CHGSP`).

In Figure 8 we have adapted the first CO-NETS specification for coping with such dynamic behaviour. More precisely, first we have to set the two transitions `ENTER` and `CHGSP` as subject to modification (i.e. as non-instantiated transitions). Following the above conceptualization, we have to replace their inscriptions by appropriate variables and reported their initial behaviour as tokens in the meta-place. We have used as variables IC_{e_i} (resp. IC_{c_i}), CT_{e_i} (resp. CT_{c_i}) and TC_{e_i} (resp. TC_{c_i}) for the transition `ENTER` (resp. `LMSP`). The corresponding behaviours of these two transitions are considered as initial or first versions and correspond to the two first (meta-)tokens in the place `Meta-place`. We note that for sake of clarity we have omitted the (meta-) messages and transitions of the meta-level that remain exactly as in Figure 7.

Besides these initial behaviours for the two transitions (now becoming optional), we propose to take into account the two wished changes, namely a speed-depending interval time for entering the region of interest and highest speed limitation. These tow new behaviour versions correspond to the third and fourth tokens in the meta-place respectively. In the new version for the transition `ENTER`, the time-interval (i.e. the last component in this tuple) is depending on the speed, and we require that the maximal speed while entering this region is to be less that 40 for instance. In the new version for the transition `CHGSP` we propose that the speed should not exceed for instance 160 when the train is elsewhere and 40 when the train is in this region of the gate; this obviously requires the test of the position attribute as indicated. But, what more important is that these initial or new behaviour are perceived as tokens so they can be changed at any time by the designer of the system through the corresponding transitions in the meta-level we have already explained.

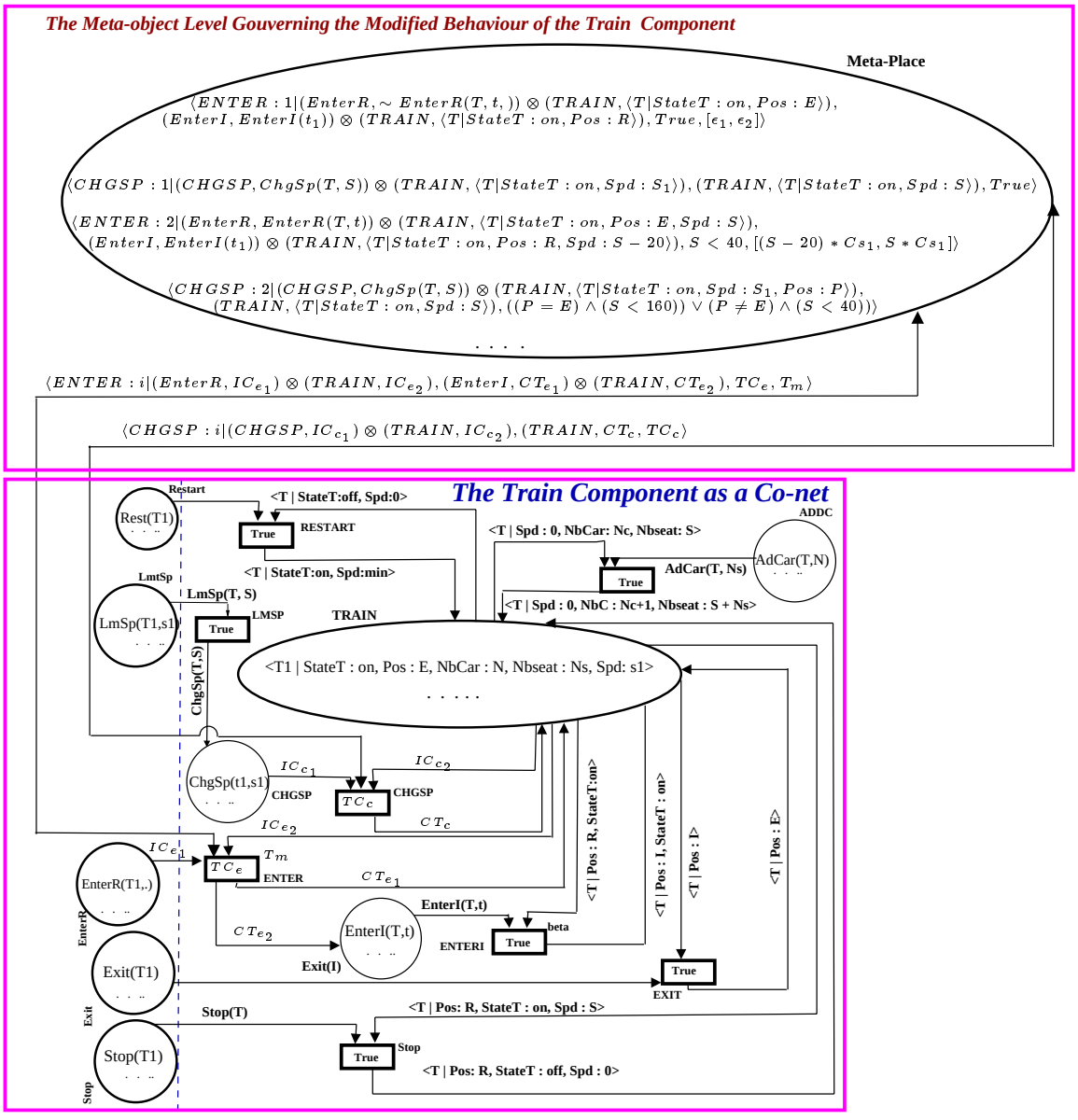


Figure 8: A runtine modifiable Co-nets train component

5 Strategies for controling CO-NETS behaviour

Besides its true concurrency nature and its operationality allowing generation of rapid-prototypes, the key feature of rewriting logic is also its *intrinsic* reflection capabilities. For our CO-NETS framework, the crucial advantage of this "second" meta-level is the possibility of introducing appropriate strategies for *controlling* the way in which different transitions should be fired. This is of great benefit for reflecting a current functioning of a given system, without resorting to fix forever a particular way of functioning. Moreover, this allows to free as much as possible the net from such control which led to a very simple and flexible net specification. In the following, we give more detail about this level, then we show how to apply it to the ETCS case study.

5.1 Strategies using reflection in rewrite logic

Rewriting logic is reflective [11], that is, there is a finitely presented rewrite theory $\mathcal{U}$ that is *universal* in the sense that we can represent in $\mathcal{U}$ any finitely presented rewrite theory $\mathcal{R}$ (including $\mathcal{U}$ itself) as a term $\overline{\mathcal{R}}$, any terms t, t' in $\mathcal{R}$ as terms $\overline{t}$, $\overline{t'}$, and any pair $(\mathcal{R}, t)$ as a term $(\overline{\mathcal{R}}, \overline{t})$, in such a way that we have the following equivalence

$$(\dagger)\ \mathcal{R} \vdash t \longrightarrow t' \iff \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \longrightarrow \langle \overline{\mathcal{R}}, \overline{t'} \rangle$$

For a system of rewrite rules gouverning transitions behaviour as in our case, it is now quite possible to represent it as (a pair of) datatypes, and the rewriting of any CO-NETS state representing the current marking can be now completely controlled. In this sense, an expressive language for composing different rewrite rules has been developed for the MAUDE language [13, 5]; that we adopt here.

First a kernel is defined stating how rewriting in the object level is accomplished at the meta-level. In particular, MAUDE supports a strategy language kernel which defines the operation:

op meta-apply : Term Label Nat -> Term.

A term **meta-apply(t,l,n)** is evaluated by converting the meta-term t to the term it represents[8] and matching the resulting term against all rules with the given label l. The first n successful matches are discarded, and if there is an $(n+1)th$ successful match its rule is applied, and the resulting term is converted to a meta-term and returned; otherwise, $error*$ is returned.

The strategy language **STRAT** defined in [5] extends the kernel with operations to compose strategies, and also with operations to create and manipulate a solution tree obtained by the application of a strategy. It defines sorts **Strategy** and **StrategyExp** for strategies, and sorts **SolTree** and **SolTreeExp** for the solution tree. The main operations defined on strategies are:

- operations defining basic strategies:

  op idle : -> Strategy . *//* idle is an empty strategy *//*
  op apply : Label -> Strategy *// application of a given rule //* .
  op rew_ =>_with_ : Term SolTreeExp Strategy -> StrategyExp .
  failure : -> StrategyExp .

- operations defining solution trees :

  op ? : -> SolTreeExp .
  op apply : Label -> Strategy .
  op rew_ =>_with_ : Term SolTreeExp Strategy -> StrategyExp .
  failure : -> StrategyExp .

- operations that compose strategies:
  op _;_ : Strategy Strategy -> Strategy *// application of two strategies in sequence //* .

[8] The data-type meta-representation of an object term is a list based one; for instance, a natural term $s(s(0)) + s(0)$ is represented by $'\_+\_['s\_['s\_['0]],'s\_['0]]$.

op _;_orelse_ : Strategy Strategy Strategy $->$ Strategy // *a choice between two strategies* //.
op iterate : Strategy $->$ Strategy // *repetitive application of a given strategy until it is no-more applied* //.

For a more detail about the semantics of these operations, the reader may particularly consult [5].

5.2 Application to the ETCS system

As we pointed out the meta-level we propose for dynamically evolving CO-NETS specification is in itself not sufficient for expressing particularly how different system transitions are performed. Indeed, first recalling that usually in Petri nets a transition may be fired as soon as it become firable, and there is no way for controlling the order in which transitions have to be fired. These difficulties induce that the designer of the system should decided whether (s)he let irrelevant such order or opt for a default order and integrate it directly in the model—leading in most of real-case to very complex Petri net which works only for this default strategy.

Obviously in specifying the ETCS problem we have decided for the first solution, that is, no control at all of different system elementary behaviour or transitions. In fact, we did not state, for instance, when should the speed be effectively limited in the train component, or when should the train enter the region of interest, etc. But imagine that we have decided for a particular fixed strategy; for instance, first limit the speed then enter into the region of interest, enter the crossing road and then exit it; and only then perform change of car numbers, or restart stopped trains, etc. It is very hard to imagine how complex and artificial would be the resulting net for such a default strategy. For instance, for firing the transition `ENTER` we have to be sure that the transitions `LMSP` and `CHGSP` have already been fired in this order. This means that we have to add an artificial output place to the `CHGSP` transition for indicating its firing, and relate this place through an artificial transition to the `EnterR` as output place.

Fortunately due to to this rewrite logic meta-reflection, the ETCS specification of different components as well as interaction remains unchanged while we can formulate any strategy we would like to have. For instance, taking into account as label the name of different transitions, the above 'default' strategy we would like to have may be simply expressed by the following:

$$iterate(LMSP; CHGSP; ENTER; ENTERI; EXIT)^*)$$

The meta-level of rewriting logic would respects this strategy. Moreover, we can associate complex conditions on applying such strategies.

6 Conclusions

We presented in this paper a general-purpose framework, referred to as CO-NETS, particularly suited for specifying / validating complex distributed object-based critical discrete-event systems, but also for dynamically manipulating their behaviour. Methodologically this framework may be regarded as three layer-based one. The first layer is a sound integration object oriented abstraction mechanisms with modularity constructs into an appropriate variety of algebraic Petri nets. The second layer is meta-level one, and it allows for dynamically creating, modifying and/or deleting any elementary behaviour while the system remain still running. The third layer takes profit of the reflection capabilities of rewrite logic—as semantics for CO-NETS behaviour—for dynamically describing as appropriate strategies reflecting the real-functioning of the complex .

We have illustrated this framework using a non-trivial specification of a variant of European train control system (ETCS). This case study demonstrates in particular the suitability of this framework for dealing complex real-world distributed systems in a very flexible. However, after achieving this first step we are conscious that much more work remains ahead. Particularly, we are planning to extend this case study variant to more complex one. Also, we are focusing on the integration of temporal aspects for verifying, and not only validating system properties.

References

[1] N. Aoumeur. Specifying Distributed and Dynamically Evolving Information Systems Using an Extended Co-Nets Approach. In G. Saake, K. Schwarz, and C Türker, editors, *Transactions and Database Dynamics*, volume 1773 of *Lecture Notes in Computer Science, Berlin*, pages 91–111. Springer-Verlag, 2000. *Selected papers from the 8th International Workshop on Foundations of Models and Languages for Data and Objects, Sep. 1999, Germany.*

[2] N. Aoumeur and G. Saake. Towards an Object Petri Nets Model for Specifying and Validating Distributed Information Systems. In M. Jarke and A. Oberweis, editors, *Proc. of the 11th Int. Conf. on Advanced Information Systems Engineering, CAiSE'99*, volume 1626 of *Lecture Notes in Computer Science*, pages 381–395. Springer-Verlag, 1999.

[3] N. Aoumeur and G. Saake. Co-nets: A Formal OO Framework for Specifying and Validating Distributed Information Systems. Preprint Nr. 2, Fakultät für Informatik, Universität Magdeburg, 2000.

[4] M. Bettaz, M. Maouche, M. Soualmi, and S. Boukebeche. Protocol Specification using ECATNets. *Reséaux et Informatique Répartie*, 3(1):7–35, 1993.

[5] M. Clavel, F. Duran, S. Eker, J. Meseguer, and M. Stehr. Maude : Specification and Programming in Rewriting Logic. Technical report, SRI, Computer Science Laboratory, March 1999. URL : http://maude.csl.sri.com.

[6] M. Clavel and J. Meseguer. Reflection and Strategies in rewriting logic. In G. Kiczales, editor, *Proc. of Reflection'96*, pages 263–288. Xerox PARC, 1996.

[7] Goguen, J. et al. Introducing OBJ. Technical Report SRI-CSL-92-03, Computer Science Laboratory, SRI International, 1992.

[8] C.L. Heitmeyer and N. Lynch. The Generalized Railroad Crossing : A Case study in Formal Verification of real-time Systems. In *Proc. of the IEE Real-Time Systems Symposium*, 1994.

[9] K. Jensen. Coloured Petri Nets: Basic Concepts, Analysis Methods and practical Use - Volume 1 : Basic Concepts. *EATCS Monographs in Computer Science*, 26, 1992.

[10] P. Kosiuczenko and Wirsing. Timed Rewriting Logic with an Application to Object-Based Specification. *Science of Computer Programming*, 28:225–246, 1997.

[11] N. Marti-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In J. Meseguer, editor, *Proc. of First International Workshop on Rewriting Logic*, volume 4 of *Electronic Notes in Theoretical Computer Science*, pages 189–224, 1996.

[12] J. Meseguer. Conditional rewriting logic as a unified model for concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[13] J. Meseguer. Solving the Inheritance Anomaly in Concurrent Object-Oriented Programming. In *ECOOP'93 - Object-Oriented Programming*, volume 707 of *Lecture Notes in Computer Science*, pages 220–246. Springer Verlag, 1993.

[14] Conrad S, J. Ramos, G. Saake, and C. Sernadas. Evolving Logical Specification in Information Systems. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*, chapter 7, pages 167–198. Kluwer Academic Publishers, Boston, 1998.

[15] E. Schneider. Referenzfallstudie Verkehrsleittechnik. Informatik-bericht, Technische Universität Braunschweig, 2000. URL : http://www.ifra.ing.tu-bs.de/ m33/spezi (In German).

[16] P. Wegner. Concepts and paradigms of Object-Oriented Programming. *OOPS Messenger*, 1:7–87, 1990.

[17] M. Wirsing and A. Knapp. A Formal Approach to Object-Oriented Software Engineering. In J. Meseguer, editor, *Proc. of the First Inter. Workshop on Rewriting Logic*, volume 4. Electronic Notes in Theoretical Computer Science, 1996.

Gate component signature:

```
obj Gate is
  protecting Object-state GRC_Structure .
  subsort Id.Gate < OId .
  subsort Local_Gate  External_Gate < Gate .
  subsort DEFECT  REPAIR  RAISE  LOWER < Obs_Gate_Mes .
```

```
  (* observed attributes *)
   op ⟨_|Position : _⟩ :
    Id.Gate  POSITION-G → External_Gate .
  (* Exported messages *)
   op Lower:  OId  Id.Gate → LOWER .
   op Raise:  OId  Id.Gate → RAISE .
   op Defect:  OId  Id.Gate → DEFECT .
   op Repair:  OId  Id.Gate → REPAIR .
endo.
```

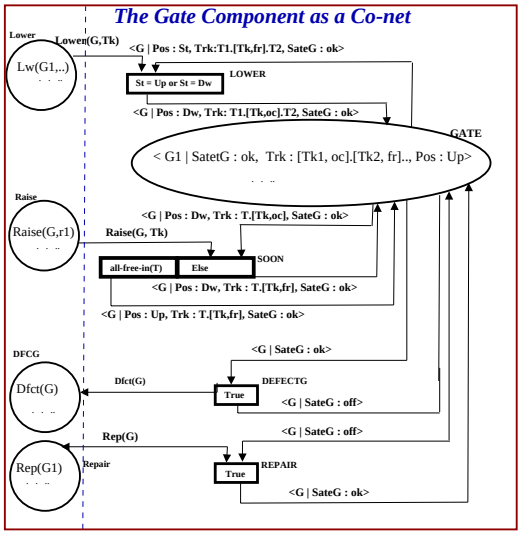


Figure 9: The Gate Component as a Co-net.

Controller component signature:

```
obj Controller is
  protecting Object-state GRC_Structure .
  subsort Id.Controller < OId .
  subsort Local_Controller  External_Controller
    < Controller .
  subsort REMVSCHD  ADDSCHD < Local_Control_Mes .
  subsort RESCHED  SOON-ARV < Obs_Control_Mes .
  (* local attributes *)
   op ⟨_|SoonAr : _, NoSoonAr : _⟩ :
    Id.Controller  bool  bool → Local_Controller .
  (* observed attributes *)
   op ⟨_|SchedTr : _⟩ :
    Id.Controller  SCHED-TR → External_Controller .
  (* local messages *)
   op Remschd :  Id.Controller  Id.Train  Id.Gate → REMSCHD .
   op Addschd :  Id.Controller  Id.Train  Id.Gate  Time → ADDSCHD .
  (* Exported messages *)
   op SonArv:  Id.Controller  OId → SOON-ARV .
endo.
```

The Light control component signature:

```
obj Light is
  protecting Object-state GRC_Structure .
  subsort Id.Light < OId .
  subsort Local_Light  External_Light
    < Controller .
  subsort G2R  R2G  DFCTL < Obs_Light_Mes .
  (* local attributes *)
   op ⟨_|atGate : _, StateL : _⟩ :
    Id.Light  Id.Gate  StateL  → Local_Light .
```

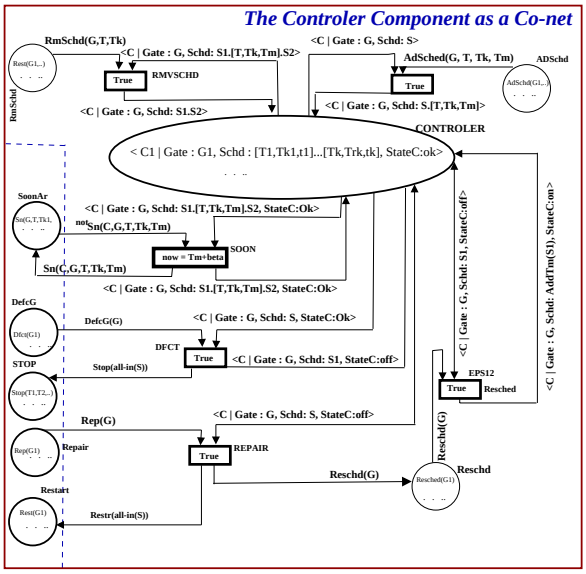


Figure 10: The Controller Component as a Co-net.

```
(* observed attributes *)
 op ⟨_|onT : _, onG : _⟩ :
  Id.Light Color Color → External_Light .
(* observed messages *)
 op G2r: Id.Light Id.Gate → G2R .
 op R2g: Id.Light Id.Gate → R2G .
 op DfctL: Id.Light Id.Gate → DFCTL .
endo.
```

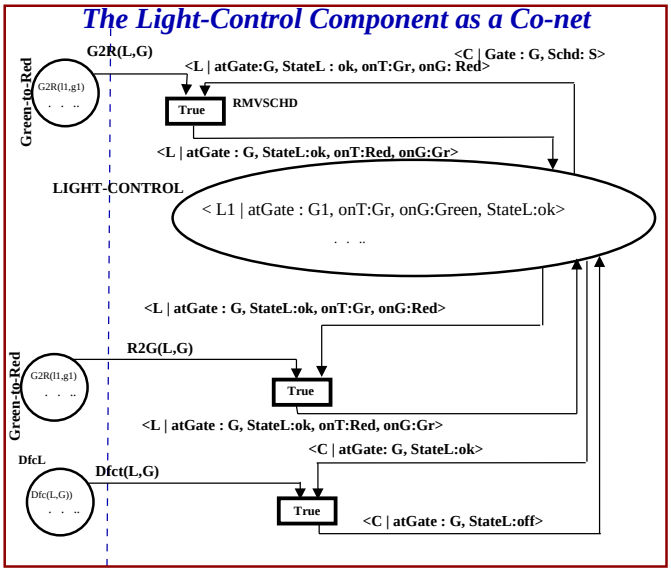


Figure 11: The Light controler Component as a Co-net.

Petri nets and components – extending the DAWN approach

Jörg Desel[1] and Ekkart Kindler[2⋆]

[1] Katholische Universität Eichstätt, Germany, joerg.desel@ku-eichstaett.de
[2] Universität Augsburg, Germany, kindler@informatik.hu-berlin.de

Abstract. The flexibility of Petri nets that allows us the combination of components without restriction to a single general composition operator is an important advantage of Petri nets compared to other formalisms. This flexibility supports many different types of component interaction, reflecting various communication paradigms. We suggest not to define the types of component interaction by adding new constructs to the language but by restricting the legal combinations between components. In this approach, general Petri net analysis techniques can still be applied to the overall model. For each component interaction type, the respective restriction is formulated in a syntactical manner, thus providing suitable *communication patterns*.
A particular communication pattern for message passing interaction is one of the core ingredients of the *DAWN* approach. DAWN was developed for modeling and verifying distributed message passing algorithms by a suitable composition of components that model components of the algorithm. In DAWN, a distributed algorithm is modeled as an *algebraic Petri net* and verified by a combination of Petri net techniques and temporal logic. The structure of a net representation as well as the verification techniques employ the particular component structure.
The DAWN approach, its underlying class of algebraic Petri nets, its communication pattern, and its verification technique is surveyed and illustrated using a simple distributed algorithm. We finally discuss how to extend the DAWN approach by other interaction paradigms, defining new communication patterns that are also based on algebraic Petri nets.

1 Introduction

Petri nets are a well-known formalism for modeling reactive and distributed systems. The following strengths of Petri nets are widely accepted:

1. Petri nets provide a precise semantics – leaving no room for ambiguities[1].
2. Petri nets provide a simple graphical notation. Visualizing the system together with its behavior and discussing it with non-experts is easily possible.
3. Petri nets come with a bunch of techniques for analyzing their behavior and for verifying their correctness. Many researchers consider these techniques the main advantage of Petri nets.

Actually, the term 'Petri net' does not denote a single formalism. Rather, it is a generic term denoting a family of related formalisms that share some common principles (see [4] for details). In this paper, the focus is on *algebraic Petri nets* [16, 14, 9], which will be discussed in Sect. 2 in some more detail.

⋆ On leave from Humboldt-Universität zu Berlin and Katholische Universität Eichstätt
[1] We admit that some engineers consider this a disadvantage in the early design phase of a system.

For large and complex systems, the second and third advantage of Petri nets listed above do not apply any more, because also the system models tend to be too large. It is generally agreed that the only way to cope with complexity is to consider components and their composition within the model. As usual for other formalisms, Petri nets can be equipped with a composition operator such that the behavior of a composed model can be derived from the behavior of its components (the so-called *principle of compositionality*). Also for Petri nets, there exist different approaches for compositional analysis or verification. Ideally, the selection of model components reflects the natural components of the modeled system and the composition operator reflects the interaction principle between the system components.

These approaches have some serious drawbacks:

- There are many different interaction principles between components of systems, e.g. message passing, synchronous communication, communication by shared variables etc., that need different composition operators.
- For each composition operator, the semantics of a formalism has to be adequately extended.
- For each composition operator, analysis and verification techniques have to be reconsidered.
- Two components can be combined only by a single operator so that the combination of different interaction principles is tedious.

In this paper, we proceed in the reverse direction. We do not extend the underlying Petri net formalism. Composition of components is generally done by an arbitrary combination of the elements of the components. For each particular interaction principle, the combination is restricted by a syntactical rule, called *communication pattern*. The main advantages of this approach are its high flexibility – arbitrary interaction principles can be allowed, and two components can be combined in different styles at the same time – and its simple semantics – the composed model is a Petri net by definition with well-defined semantics, and all Petri net techniques can be applied.

In order to manage the complexity of components and their interaction, communication patterns must be carefully designed. Clearly, a communication pattern must faithfully reflect the respective interaction principle of the system. Moreover, the resulting net model and in particular its graphical representation should be easy to understand. It must be simple to identify single components and their interaction. Finally, analysis and verification techniques should be able to employ the component structure. This way, the inherent complexity of these techniques can be reduced whenever a suitable decomposition is known.

It turns out that this approach works well for modeling and verifying distributed systems and algorithms that are based on the message passing paradigm with algebraic Petri nets. The approach called *DAWN* (the Distributed Algorithms Working Notation) [10, 3, 18, 5, 12, 13] uses a particular communication pattern for message passing and provides powerful verification techniques. In order to justify our claim, we give a brief survey on the basic concepts of DAWN (Sect. 2). Its verification technique is demonstrated for a simple distributed algorithm.

Finally, we discuss the communication pattern used for modeling distributed algorithms (Sect. 3.2) and present new communication patterns that represent other communication paradigms (Sect. 3.3–3.5). As the DAWN-approach and its communication pattern is based on algebraic Petri nets, so are the new communication patterns.

2 DAWN

In this section, we give an overview on DAWN. The basic concepts of DAWN are discussed in Sect. 2.1. These concepts are illustrated by the help of an example in Sections 2.2–2.4. Moreover, there will be brief discussion on tool support for DAWN (Sect. 2.5).

2.1 Concepts

Distributed algorithms. DAWN is designed for modeling and verifying distributed algorithms. A *distributed algorithm* is executed on a network of independent computation devices, which can communicate with each other by exchanging messages. We call a computation device a *component*[2]. Each component has its local state, which cannot be accessed by other components. The only way of interaction is sending and receiving messages over cannels of an underlying *communication network*. Usually, a distributed algorithm is not based on a fixed topology. Rather, any topology satisfying some specification can be used. Some distributed algorithms work on a directed ring, others work on a tree or on a fully connected graph. The topology can even change during the run of a distributed algorithm.

Typically, the components execute identical local algorithms. However, the respective local states as well as the neighbors in the communication network may be different for each agent. In order not to confuse the local algorithm executed by a component with the distributed algorithm of all components, we call the local algorithm a *protocol* in the rest of this paper. Altogether, a distributed algorithm is given by

1. the protocol executed by each component,
2. initial local states for each component, and
3. a specification of the underlying communication topology.

Each communication network satisfying the specification constitutes an *instance* of the distributed algorithm. Typically, a distributed algorithm has infinitely many instances.

High-level nets and algebraic nets. In DAWN, a distributed algorithm is modeled as an algebraic Petri net. An instance of a distributed algorithm is modeled by a high-level Petri net. These concepts will be introduced below. For technical details of high-level Petri nets see [6–8] and of algebraic Petri nets see [9].

In contrast to low-level Petri nets, *high-level Petri nets* have distinguishable token. For each place, a *token domain* specifies the legal tokens for that place. A marking of a place is a multiset over its token domain. In a high-level Petri net, a transition can fire in different *modes*. The possible modes of a transition are defined using a set of *transition variables*, each of which is associated with a domain defining its legal values. Each assignment of legal values to the transition variables represents a mode of the transition. When a transition fires in a mode, it consumes tokens from its input places and produces tokens on its output places. Which tokens are consumed and which tokens are produced is defined by the inscription of the corresponding arcs. An *arc inscription* is an expression, in which transition variables may occur. The tokens consumed or produced by the occurrence of a transition in a mode are given by the evaluation of this expression, where the transition variables take the respective values given by the mode.

In high-level Petri nets, the syntax and the semantics for the token domains and for the arc inscriptions are fixed. Formulated in algebraic Petri net terminology, the symbols used for token domains and for arc inscriptions are associated with a fixed algebra, which canonically defines how to evaluate the expressions. In algebraic Petri nets, the semantics of the token domains and the arc inscriptions is not fixed. An algebraic Petri net comes with a class of algebras. Fixing an algebra from this class gives us an *instance* of the algebraic Petri net[3], a high-level net.

[2] A component is sometimes called an *agent* of the distributed algorithm. The term agent, however, suggests some kind of intelligence, which does not apply here. Components are 'stupid' computation devices executing their part of the distributed algorithm.

[3] In order to stress the fact that an algebraic Petri net has different instances, algebraic nets are called *algebraic net schemes* in [9, 13].

Modeling distributed algorithms. The relation of an algebraic Petri net to its instances is analog to the relation of a distributed algorithm to its instances. We exploit this analogy in the following way: As stated above, a distributed algorithm is modeled as an algebraic Petri net. In this algebraic Petri net, some symbols concerning the communication topology remain without a fixed semantics. For example, the symbol A stands for the nodes of the communication network, the symbol $N \subseteq A \times A$ stands for the neighborhood in the communication network, and $N(x)$ stands for the set[4] of all neighbors of x. When fixing a communication network (out of the communication topology of the distributed algorithm), we fix the interpretation of these symbols accordingly. In this way, there is an exact correspondence between the instances of the algebraic Petri net and the instances of the distributed algorithm.

2.2 An example: Modeling

In order to illustrate the concepts above, let us consider an example: a simple token ring algorithm. Its task is to allow any component of a distributed system to repeatedly require access to its critical section and then enter its critical section whereas never more than one component should be in its critical section at the same time.

Consider the Petri net in Fig. 1. Three components, a, b, c of a distributed system are cyclically connected. There exists one token that is passed from component to component in this ring. Initially, the token is with a. Each component can be idle – then it will just pass on the token after receipt. An idle component can become requesting. Then, it will enter its critical section as soon as it receives the token. When leaving the critical section, the component becomes idle again.

Correctness of this algorithm concerns two properties:

(A) At no time, two components are in their critical section.
(B) If a component ever requests access to its critical section, then it will eventually enter its critical section after this request.

Experts might recognize that the second property cannot be guaranteed without some further assumptions. First, it is important to assume that no component can boycott the algorithm by refusing any action. All transitions except $a.2, b.2, c.2$ enjoy a progress assumption; they are not allowed to remain enabled forever without occurring. Second, one could argue that the transition $a.2$ moving to the requesting state of $a.2$ competes with the transition $a.1$ provided both transitions are enabled (and similarly for the other components). To cope with this problem, some fairness assumption would be necessary. For simplicity sake, we ignore this subtle problems and go on without fairness, despite the fact that mutual exclusion algorithms formulated properly can be proven to work only in the presence of a fairness assumption [11].

Instead of proving the correctness of the elementary net, we proceed with modeling techniques. We extend the previous example by a counter that counts the number of accesses to the critical section. To this end, the token could be represented as a natural number that increases whenever a component enters is critical section (see Figure 3). This net is already a high-level Petri net. It has infinitely many reachable markings because the counter variable increases beyond any bound. A low-level net unfolding of this net is hard to draw; it would demand infinitely many copies of the places $in, critical, out$ and of the transitions $1, 3, 4, 5$ for each component – one for each natural number n.

Folding of the three components of the low-level net of Figure 1 yields the high-level net of Figure 2, where the symbol A stands for the (multi)set $\{a, b, c\}$. Likewise, folding of the high-level net of Figure 3 yields the net of Figure 4, which can be considered high-level in the two dimensions data and topology.

[4] Actually, $N(x)$ technically does not denote a set, but a multiset.

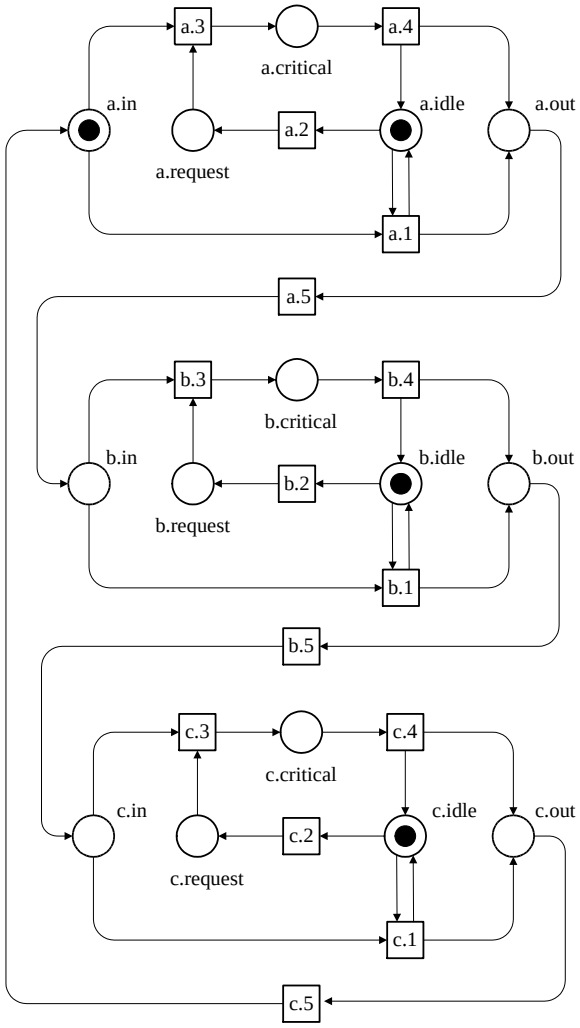


Fig. 1. The algorithm as an elementary Petri net

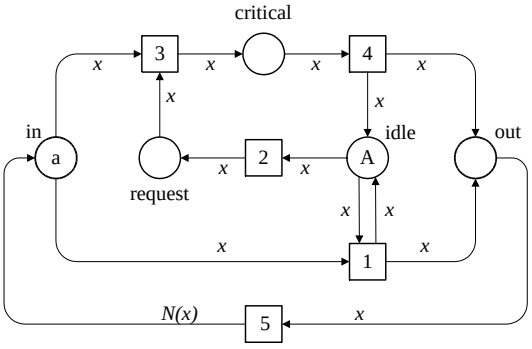


Fig. 2. A high-level net representation of the algorithm

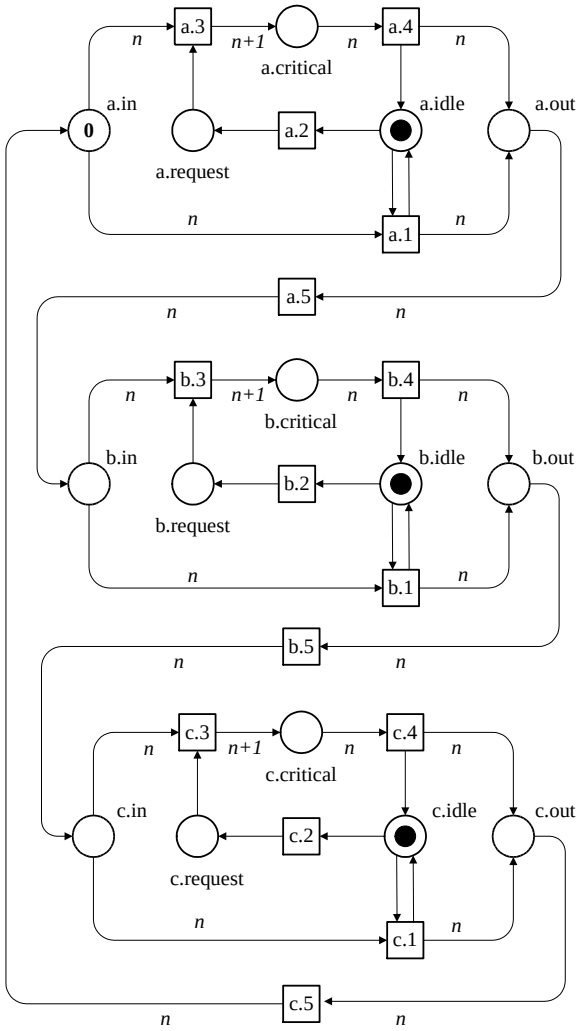


Fig. 3. The net of Figure 1 supplemented with a counter

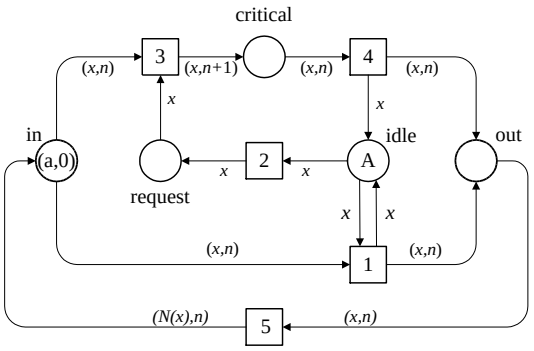


Fig. 4. A high-level net representation with counter

In both representations, the arc annotation x can be viewed as a variable for the component. If a transition occurs in some mode, then all variables at adjacent arcs are consistently assigned a value. In the example of Fig. 2, only x has to be given a value of the set $\{a, b, c\}$. Initially, transition 1 is enabled for the assignment a and transition 2 is enabled for all three assignments. $N(x)$ denotes the next neighbor[5] in the ring for each component x. So $N(a) = b$, $N(b) = c$ and $N(c) = a$. In the example of Fig. 4, some tokens and, correspondingly, some arc annotations are pairs. The first component plays the role of the component, the second of the data value (see Sect. 3.1 for a more detailed discussion).

Up to now, we have considered two models of single instances of two different distributed algorithms. In both cases, the underlying communication network was a directed ring with the three nodes a, b and c. Of course, the distributed algorithm works also for a ring with four or five nodes. In fact, it works for any directed ring. Next, we show how to represent a complete distributed algorithm (not only a single instance of it). Surprisingly, we do not need any further Petri net picture. We just switch from high-level nets to algebraic Petri nets.

Consider again the high-level net of Figure 2. In contrast to the above explanation, we do not interpret the symbol A by the fixed set anymore. Now, it can be any set. Since this algorithm depends on the assumption that one component possesses the token initially, we should require that a is an element of A. Moreover, it is essential that the token can and, eventually, will reach any component. This is guaranteed because the network topology is a directed ring. The topology is represented by the symbol N. So we must specify that N defines a relation on A such that its graph is a directed ring, no matter what A looks like. So formally, the algebraic net is a specification of an infinite set of high-level nets. At the same time, this net can be viewed as a parameterized high-level net, taking a, A and N as parameters. As mentioned above, these parameters must meet some restrictions. In our example, these requirements are:

$a \in A$,
$\forall x \in A : \mid N(x) = 1 \mid$, and
$\forall x, y \in A : \exists n : N^n(y) = x$.

The second requirement states that each component has exactly one successor. The third requirement states that each component x is reachable from any other component y in the communication network. The set of high-level nets obtained by a correct assignment of the parameters coincides with the set of models satisfying the specification of the algebraic net. So we do not need any new theory for parameterized high-level net – they are algebraic nets.

Finally, the net of Figure 4 can be viewed as a parameterized high-level as well. Here we have more free parameters: the set A, its element a and a suitable relation N as above, and moreover any interpretation of the symbols 0 and +. Taking the initial semantics of an algebraic interpretation, the natural numbers are obtained. However, other semantics work as well.

In the following sections it will be shown how the correctness of all possible interpretations, i.e. of all high-level nets satisfying the specification given by an algebraic net, can be proven.

2.3 An example: Specification

In Sect. 2.4, we illustrate how to verify the above algorithm in DAWN. Before, we must formalize the specification of the algorithm.

First, we consider the mutex property, which states that never two components are in their critical section at the same time. In our Petri net model, a component x is in its critical section

[5] In general, $N(x)$ denotes the set of all neighbors of x. In a directed ring, however, each component has exactly one neighbor: its successor. Therefore, $N(x)$ is a multiset with exactly one element for each component x.

if there is a token x on the place *critical*. Thus, the multiset of tokens on the place *critical* represents all components that are in their critical section at that state. Consequently, the mutex property can be expressed by the following temporal formula:

$$(A) \quad \Box \mid critical \mid \leq 1$$

The temporal operator $\Box$ ('always') states that the state formula $\mid critical \mid \leq 1$ holds true in all reachable states of the Petri net model. In a given state, a place name in a state formula represents the multiset of tokens on that place. Altogether, the formula states that there is always at most one component in its critical section.

The second property states that each component requesting access to the critical section will eventually enter its critical section. This can be expressed by the following temporal formula

$$(B) \quad request(x) \rightsquigarrow critical(x)$$

The temporal operator $\rightsquigarrow$ ('leadsto') states that, in each execution of the Petri net model, the following property holds: Whenever there is a state in which $request(x)$ holds true, there will be a state later in that execution in which $critical(x)$ holds true. The state formula $request(x)$ means that there is at least one token x on the corresponding place *request* in a given state, and similarly for $critical(x)$. Altogether, the formula (B) states that each requesting component x will eventually get access to its critical section.

2.4 An example: Verification

DAWN [18, 12, 13] combines verification techniques from Petri net theory with techniques from temporal logic. The proofs are designed by hand, but there are techniques that allow us to fill in missing details and to check these proofs fully automatically [2].

In order to give a flavor of these proof techniques, we give a proof for the above example. We will introduce all notations and techniques where necessary in the proof. A more concise introduction to DAWN and a presentation of its theoretical foundations can be found in [18, 12, 13, 2]. The proofs of properties (A) and (B) can be found in Tables 1 and 2, respectively. The proofs consists of lines were each line has three parts: a line number, the proven property, and a proof argument. In these proofs, we abbreviate *critical* by *crit* and *request* by *req*, in order to have a more compact representation of the formulas.

Let us discuss these proofs in some more detail. The proof of property (A) in Table 1 is quite simple: Line (1) says that there is always exactly one token on one of the places *crit*, *in* or *out*. This property can be proven[6] from a place invariant of the Petri net model. The property of line (2) is an immediate consequence of the property proven in line (1); this is indicated by the argument 'Weakening'.

(1) $\Box \mid in + crit + out \mid = 1$	Place invariant
(2) $\Box \mid crit \mid \leq 1$	Weakening (1)

Table 1. A proof of property (A)

The proof of property (B) in Table 2 is more involved. Line (3) is another weakening of property (1): From (1), we know that there is exactly one token on one of the places *in*, *crit*

[6] In fact, this could be checked fully automatically by automated theorem provers for this and the following lines [2].

and *out*. In line (3), we express this by the existence of some token y. Line (4) states that the multiset of tokens on the places *idle*, *req* and *crit* is the set A (i.e. the multiset in which each element of A occurs exactly once). Intuitively, this reflects the fact that each component is in exactly one of the states *idle*, *req*, or *crit*. Formally, this can be proven by a place invariant of the Petri net again. This property is weakened in line (5). On the right hand side of the implication, we have a combination of the left hand side with the property from line (4), with some simplifications.

(3) $\Box\ \exists y \in A : in(y) \vee crit(y) \vee out(y)$	Weakening (1)
(4) $\Box\ idle + req + crit = A$	Place invariant
(5) $\Box\ (in(y) \vee crit(y) \vee out(y)) \Rightarrow$ $(in(y) \wedge idle(y) \vee in(y) \wedge req(y) \vee crit(y) \vee out(y))$	Weakening (4)
(6) $(in(y) \wedge idle(y)) \leadsto (in(y) \wedge req(y) \vee crit(y) \vee out(y))$	Progress: 1
(7) $(in(y) \wedge req(y)) \leadsto (crit(y) \vee out(y))$	Progress: 3
(8) $crit(y) \leadsto out(y)$	Progress 4
(9) $out(y) \leadsto in(N(y))$	Progress: 4
(10) $(in(y) \vee crit(y) \vee out(y)) \leadsto in(N(y))$	Proof graph 1
(11) $(in(y) \vee crit(y) \vee out(y)) \leadsto in(N^n(y))$	Induction: (10)
(12) $\exists n \in nat : x = N^n(y)$	Assumption (ring)
(13) $(in(y) \vee crit(y) \vee out(y)) \leadsto in(x)$	Subst.: (12) in (11)
(14) $\Box\ req(x) \Rightarrow (req(x) \wedge \exists y \in A(in(y) \vee crit(y) \vee out(y)))$	Weakening (3)
(15) $req(x)$ waitsfor $in(x)$	Waitsfor
(16) $(req(x) \wedge \exists y \in A : (in(y) \vee crit(y) \vee out(y))) \leadsto$ $(req(x) \wedge in(x))$	$\leadsto$/synch (13), (15)
(17) $(req(x) \wedge in(x)) \leadsto crit(x)$	Progress: 3; 1 excl. by (4)
(18) $req(x) \leadsto crit(x)$	Transit. (14), (16), (17)

Table 2. A proof of property (B)

In lines (6)–(9) we prove some simple leadsto properties that, basically, reflect the occurrence of one transition of the Petri net. This is indicated by the argument 'Progress' along with the number of the involved transition. Due to conflicts, however, it could be that in the state indicated on the left hand side of the leadsto property another transition than the one mentioned in the proof argument occurs. In that case, also this other transition must guarantee that the right hand side of the leadsto property holds true after its occurrence. Let us discuss this for line (6) in some more detail. In a state satisfying $in(y) \wedge idle(y)$ transition 1 is enabled (with x bound to y). If this transition occurs, it moves a token y to place *out*; thus, $out(y)$ is satisfied after the occurrence. However, there are two other transitions that could be in conflict with transition 1 in this particular mode: transition 2 and transition 3 in mode $x = y$. The occurrence of transition 2 results in a state $in(y) \wedge req(y)$; the occurrence of transition 3 results in a state $crit(y)$. In either case, the right hand side of the leadsto property is satisfied. The arguments for lines (7)–(9) are quite similar. Checking the details is quite tedious; but, again, this can be checked fully automatically.

In line (10), we combine the already proven leadsto property to a more complex one. We start in a state satisfying the left-hand side of the leadsto property. This state is shown on the left-hand side of the proof graph 1 in Table 3. By property (5) we know that we are in one of the states $in(y) \wedge idle(y)$, $in(y) \wedge req(y)$, $crit(y)$, or $out(y)$ – one of the disjuncts on the right-hand side of the implication of (5). In the proof graph this is indicated by the arcs towards the corresponding disjuncts. The label (5) refers to the argument for these arcs. Likewise the arcs from the other states of the proof graph are justified by leadsto properties (with a corresponding

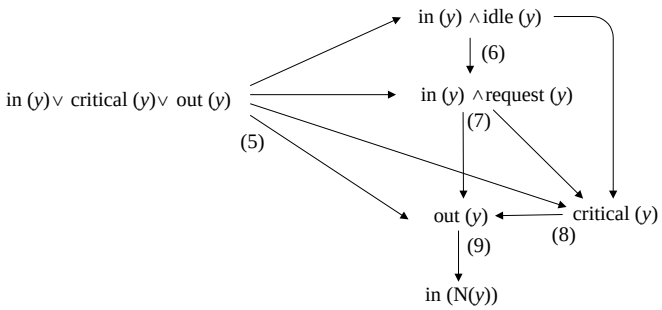


Table 3. Proof graph 1

disjunct on the right-hand side). Altogether, the proof graph shows that we eventually end up in a state satisfying $in(N(y))$; the proof graph (justified by lines (5)–(9) is the argument for the property in line (10).

Line (11) is an inductive application of line (10). Line (12) states that each component x is the n^{th} neighbor of y for some n. This was the requirement on the communication topology. Combining this property with (11), we know that, if there is some token y somewhere on one of the places *in*, *crit*, or *out*, there will eventually be a token x on place *in* for each x (line (13)).

At last, we show that a request will eventually result in an access to the critical section. In line (14) we use property (3) another time: we add it to the right-hand side of the implication. Technically, this is a weakening of (3) again. Next we know that no transition can remove a token x from place *req*, as long as *in* has no token x. This can be immediately checked from the corresponding net. Only transition 3 can remove a token x from place *req*. But this transition needs also a token x on place *in*. In combination with (13), we know that we have a token x on places *in* and *req* eventually (line (16)). From such a state, the occurrence of transition 3 will establish $crit(x)$. Note that transition 1 is statically in conflict with transition 3; however from invariant (4), we know that transition 1 cannot occur as long as transition 3 is enabled. Combining lines (14),(16), and (17) transitively gives us property (B) from the specification.

2.5 Tool support

In the previous section, we have proved that the simple token ring algorithm works for a ring of arbitrary size. Note that we have shown safety properties as well as liveness properties of the algorithm. Up to now, a DAWN proof is always designed by hand. This has advantages and disadvantages.

The advantage is that the proof does not only tell that the algorithm is correct (which would be the answer of a modelchecker in the case of correctness); the proof provides also some clue why the algorithm is correct, and it identifies the basic arguments.

The disadvantage is that it may be quite hard and time-consuming to design a proof for a larger system completely by hand. Even checking the validity of a proof may be quite tedious. In order to deal with this problem, we have proposed two automatic techniques that support the design and the checking of a DAWN proof. In [2], we have shown that automated theorem provers can check the correctness of a proof fully automatically. Indeed, this technique can be used to 'guess' a proof without being really sure; then the theorem prover can be used to check this proof. If the proof is not yet correct, the missing arguments identified by the theorem prover, could help to correct to proof. At least, this technique frees us from the most tedious task: checking all the details. A prototype of this tool was implemented within a diploma thesis [15].

The other technique is modelchecking. When designing a proof for some algorithm, we can never be sure that we will eventually find a proof – for the simple reason that the algorithm might be incorrect. In order to identify incorrectness of an algorithm, we use classical modelchecking. Since the algorithm is modeled as an algebraic net, we can easily produce some high-level instances of it (in our example, we could simply produces the instance for a ring with 5 components). Since these instances are typically finite, we can do classical modelchecking. So we can automatically generate different instances of the algorithm and start a modelchecker. If a modelchecker finds an instance that does not meet the requirement, we know that the algorithm is not yet correct. So, we would better correct the algorithm, before restarting the whole procedure. A prototype of this tool was implemented within another diploma thesis [1].

In principle, both techniques are independent of each other. Theorem proving helps us proving correctness, modelchecking helps us proving incorrectness. Thus they nicely complement each other and should be used in combination: While trying to find a proof with theorem prover support, an instance generator along with a modelchecker should be run in parallel in order to make sure that we do not work on a proof of a faulty algorithm.

Combining an automated theorem prover with a modelchecker gives us a semi-automatic

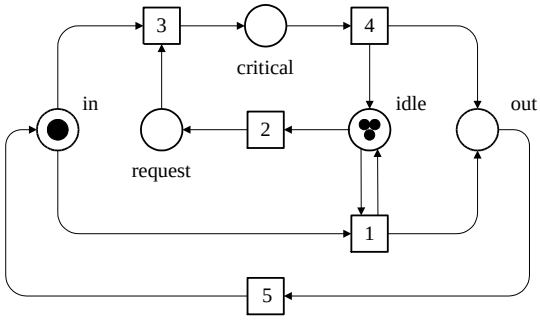


Fig. 5. The skeleton

tool for verifying parameterized distributed systems. In some cases, we can do even better. Property (A), for example, can be proven fully automatically by considering the *skeleton* of the algebraic net. Figure 5 shows the skeleton of our algebraic Petri net from Fig. 2. Basically, we have omitted the inscriptions of the algebraic net and converted the individual tokens to single black tokens; this gives us a classical low-level Petri net. Obviously, the skeleton satisfies the property $\Box \mid \mathit{critical} \mid \leq 1$. This property follows from a classical place invariant of the skeleton and, therefore, could be proven fully automatically. Due to some syntactical restrictions, which we will not discuss in detail here, the property of the low-level net also holds true for the algebraic net. Therefore, property (A) could be proven fully automatically in our example.

At least in some cases, skeletons can be used to verify properties of algebraic nets fully automatically. Most times, this applies to invariants. Sometimes, however, it applies even to liveness properties. A detailed investigation of this issue, however, is subject to future research.

2.6 Summary

In this section, we have illustrated the use of DAWN for modeling and verifying distributed algorithms. The protocols of the components as well as their interaction are modeled within a single algebraic Petri net. This results in a concise model of the distributed algorithm, without

any notational or technical overhead. The model captures the *algorithmic idea* [17, 13] of the distributed algorithm.

The resulting models are – in most cases – small enough for formal analysis and verification.

3 Designing Petri net models

In Sect. 2, we have shown that a distributed algorithm using message passing communication can be modeled as an algebraic Petri net. Communication between components is modeled by transition 5, which sends a message from an agent to its neighbor. This sending operation is reflected by the arc annotation $N(x)$ at an outgoing arc of the transition. It will be shown that this kind of arc annotations obeys the rule of a certain communication pattern.

We start this section with the general design principle of DAWN and its communication pattern (message passing). Then, we will discuss some more general communication patterns.

3.1 The design principle

Before presenting the design principle of DAWN, let us ask the following question: Is every algebraic Petri net a model of some distributed algorithm (in the sense defined in Sect. 2.1)?

The answer to this question is a clear 'no'! There are algebraic Petri nets that do not model a distributed algorithm – not even a stupid one. For example, there could be a transition that accesses the local states of two different components – violating our requirement on distributed algorithms. This is exactly what our design principle should exclude.

In DAWN, accessing the local states of two different components is syntactically excluded in the following way: Each token of the Petri net can be considered as a tuple[7]. Now, we assume that the first element of the token (tuple) is of type A – i. e. it represents some component of the underlying communication network. Furthermore, we assume that a token with first element a indicates that the token belongs to the local state of component a. Therefore, each token can be uniquely associated with a component of the underlying communication network. We call this component the *owner* of the token.

Now, we give a syntactical restriction for transitions that access the local state of a single component [3]: all inscriptions of the adjacent arcs of this transition must be of the form $(x, ...)$, where x is a variable of type A. This way, we know that all tokens consumed or produced by this transition are owned by the same component x (or rather the value assigned to x in the particular mode). We say that the transition is *local*. Transitions 1–4 in Fig. 2 and in Fig. 4 meet this requirement. We will explain in a minute why transition 5 does not meet this requirement.

Altogether, the underlying design principle associates each token with a component, its owner. We have chosen to represent a token as a tuple and use its first element as its owner. Of course, we could have chosen any other element of the tuple to represent the owner. But, the first element turned out to be a sensible choice. A transition is local if all tokens consumed and produced by an occurrence of this transition belong to the same owner.

Of course, not all actions of a distributed algorithm are local because of interactions between the components. For the moment, we call all non-local transitions *interactions*. In our example, transition 5 is an interaction transition. Characterizing the legal interactions, however, depends on the communication paradigm. Therefore, the syntactical restrictions for these interaction transitions depend on the communication paradigm. We call such restrictions communication patterns. Some communication paradigms along with their communication patterns are discussed below.

[7] If it is not a tuple, we can consider it as a 1-tuple.

3.2 Message passing

For distributed algorithms, the communication paradigm is message passing. In a Petri net model, sending a message to some component y can be modeled by putting a token with first element y on some place. Therefore, we do not restrict the inscriptions of the out-going arcs of such a transition. The inscriptions of the in-coming arcs, however, must have the form $(x, ...)$. We call transitions with this restriction *message passing* transitions.

Transition 5 in Fig. 2 and in Fig. 4 is a message passing transition. The only in-coming arc has the inscription x (a 1-tuple with first element x). For the out-going arc, there is no restriction ($N(x)$ denotes the neighbor of x; thus x puts a token to the place *in* of its neighbor).

3.3 Synchronous communication

Next, we consider synchronous communication: a joint action of two[8] components x and y. Since it is a joint action, the action may access and change the local state of both components x and y. We call the corresponding transitions *synchronization transitions*.

Syntactically, a synchronization transition can be characterized as follows: Each arc-inscription is of the form $(x, ...)$ or of the form $(y, ...)$, where x and y are variables of type A.

Note that we do not require that a synchronization transition has in-coming arcs of both forms $(x, ...)$ and $(y, ...)$. So, a synchronization transition can be also a local transition or a message passing transition. If necessary, this can be easily excluded by an additional requirement.

Synchronous communication, however, is kind of magic. Two completely independent components x and y meet each other by chance and, then, execute a joint action. Surly, this is the right level of abstraction for some applications. For others, we would like to guarantee, that components do meet in a controlled way. For example, a telephone call is such a controlled synchronous interaction: one partner dials the number of the other partner. Here, we give an additional restriction that reflects this controlled synchronization: an additional transition guard $y = exp$, where exp is an expression in which only variables of in-coming arcs of the form $(x, ...)$ occur. This restriction guarantees, that x knows how to contact y (in a sense, x looks up the telephone number of y in its local state).

3.4 Shared variables

Often, components communicate over a set of shared variables. The set of all shared variables can be represented as tokens on a distinguished place, where each shared variable is represented by a pair. The first element of the pair represents the name of the variable, the second element represents its current value.

Now, an access to a shared variable can be modeled by a transition with an arc to and from this place where the arc-inscriptions have the form $(v, ...)$ (the transition variable v denotes the name of the accessed variable). In a read access, both inscriptions must be (v, q), where q is a transition variable denoting the value of the variable, which is not changed by a read access. Note, however, that the usual Petri net semantics does not allow concurrent read operations on one variable modeled this way. In a write access, the inscription of the arc leading to the transition should be (v, q') and the inscription of the arc leading to the place should be (v, q) where q' is a transition variable that does not occur in any other inscription of this transition (the old value q' of the variable is lost).

[8] Here, we consider synchronous communication of two components only. A generalization to multi-party-interactions, however, is straightforward.

3.5 Rendezvous

A rendezvous is 'a place where people meet'. In our context, it is a place, where components meet for common interactions. In order not to confuse Petri net places with places for rendezvous, we call places for rendezvous *locations* in the following.

A rendezvous location can be represented by another distinguished Petri net place: a *rendezvous place*. The tokens on this place are pairs, where the first element of the pair denotes a rendezvous location, and the second element denotes a multiset of components (the components meeting at this place).

Each component may register itself with a rendezvous location and unregister itself. The corresponding transitions must have an arc to and from the rendezvous place with arc-inscriptions $(l, L + x)$ and (l, L) where l is a variable for locations and L is a variable for multisets of components. If the arc to the transition carries the first inscription then the transition adds x to the rendezvous, otherwise it removes x.

A *rendezvous transition* is a transition with a loop to the rendezvous place with arc-inscription (l,L), where l denotes the rendezvous location and L the multiset of components currently meeting at this location. There are no restrictions for the other arcs of a rendezvous transition. However, a rendezvous transition has additional transition guards: for each variable z occurring in the first place of an arc-inscription there is a transition guard $z \in L$. This guarantees, that the rendezvous transition accesses only local states of components that participate in the corresponding rendezvous.

4 Conclusion

In this paper, we claim that algebraic Petri nets in combination with communication patterns help us to develop concise Petri net models of distributed systems. Instead of extending Petri nets with composition operations along with sophisticated compositional semantics, we restrict the use of algebraic Petri nets by simple communication patterns.

In order to motivate and justify this claim, we have surveyed the DAWN approach, its underlying concepts and design principles. Its main features are:

1. The protocols executed by the different components are folded onto the same Petri net.
2. The use of algebraic Petri nets, which precisely reflect the relation between distributed algorithms and their instances. This makes algebraic Petri nets a superb candidate for modeling distributed algorithms.
3. Modeling interaction by communication patterns instead of explicit composition operations avoids technical overhead and allows us to focus on the algorithmic idea.

References

1. Abdourahaman. Model checking in DAWN. Master's thesis, Humboldt-Universität zu Berlin, Institut für Informatik, September 2000.
2. Thomas Baar, Ekkart Kindler, and Hagen Völzer. Verifying intuition – ILF checks DAWN proofs. In S. Donatelli and J. Kleijn, editors, *Application and Theory of Petri Nets 1999, 20^{th} International Conference*, *LNCS* 1639, 404–423. Springer, June 1999.
3. Jörg Desel. How distributed algorithms play the token game. In *Foundations of Computer Science: Potential – Theory – Cognition*, *LNCS* 1337. Springer, 1997.
4. Jörg Desel and Gabriel Juhas. 'What is a Petri net' – informed answers for the informed reader. In H. Ehrig, G. Juhas, J. Padberg, and G. Rozenberg, editors, *Unifying Petri Nets, Advances in Petri Nets*, *LNCS* 2128, 1–27. Springer, 2001.

5. Jörg Desel and Ekkart Kindler. Proving correctness of distributed algorithms using high-level Petri nets – a case study. In *1998 International Conference on Application of Concurrency to System Design*,177–186, Fukushima, Japan, March 1998. IEEE Computer Society Press.
6. Hartmann J. Genrich and Kurt Lautenbach. System modelling with high-level Petri nets. *Theoretical Computer Science*, 13:109–136, 1981.
7. Kurt Jensen. Coloured Petri nets and invariant methods. *Theoretical Computer Science*, 14:317–336, 1981.
8. Kurt Jensen. *Coloured Petri Nets, Volume 1: Basic Concepts*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1992.
9. Ekkart Kindler and Wolfgang Reisig. Algebraic system nets for modelling distributed algorithms. *Petri Net Newsletter*, 51:16–31, December 1996.
10. Ekkart Kindler and Wolfgang Reisig. Verification of distributed algorithms with algebraic Petri nets. In C. Freksa, M. Jantzen, and R. Valk, editors, *Foundations of Computer Science: Potential – Theory – Cognition*, *LNCS* 1337, 261–270. Springer-Verlag, 1997.
11. Ekkart Kindler and Rolf Walter. Mutex needs fairness. *Information Processing Letters*, 62:31–39, 1997.
12. W. Reisig, E. Kindler, T. Vesper, H. Völzer, and R. Walter. Distributed algorithms for networks of agents. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets II: Applications*, *LNCS* 1492, 331–385. Springer, 1998.
13. Wolfgang Reisig. *Elements of Distributed Algorithms — Modeling and Analysis with Petri Nets*. Springer, 1998.
14. Wolfgang Reisig and Jacques Vautherin. An algebraic approach to high level Petri nets. In *Proceedings of the VIII European Workshop on Application and Theory of Petri Nets*, 1987.
15. Sebastian Unger. Automatisches Überprüfen von DAWN-Beweisen. Master's thesis, Humboldt-Universität zu Berlin, Institut für Informatik, September 1999.
16. Jacques Vautherin. Parallel systems specifications with coloured Petri nets and algebraic specifications. In G. Rozenberg, editor, *Advances in Petri Nets*, *LNCS* 266, 293–308. Springer-Verlag, 1987.
17. Rolf Walter. *Petrinetzmodelle verteilter Algorithmen – Beweistechnik und Intuition*. PhD thesis, Humboldt-Universität zu Berlin, Institut für Informatik, 1995.
18. M. Weber, R. Walter, H. Völzer, T. Vesper, W. Reisig, S. Peuker, E. Kindler, J. Freiheit, and J. Desel. DAWN: Petrinetzmodelle zur Verifikation Verteilter Algorithmen. Informatik-Bericht 88, Humboldt-Universität zu Berlin, December 1997.

AN APPLICATION OF AN EXPRESSIVE COLOURED PETRI NETS MODELING METHODOLOGY TO A BUSINESS TO BUSINESS ENVIRONMENT

JUAN FRAUSTO-SOLIS[1], FRANCISCO CAMARGO-SANTACRUZ[2] and FERNANDO RAMOS-QUINTANA[1]

Instituto Tecnológico y de Estudios Superiores de Monterrey, [1]Campus Cuernavaca, [2]Campus Estado de México, Km. 3.5, Carretera al Lago de Guadalupe, Atizapán, 52926, Estado de México, México, phone ++52-5864 5560, fax ++52-5864 5557, http://www.itesm.mx.

E-mail: [1]{framos, jfrausto} @campus.mor.itesm.mx, [2]fcamargo@campus.cem.itesm.mx

The dynamic nature of cooperative agent environment makes considerably more difficult the task of modeling permanent interactions among agents. This problem becomes rather hard if more than two agents are involved. So far, traditional approaches deal with the problem of modeling interactions in static conditions and commonly with only two agents participating concurrently in cooperative tasks. The complex nature of such dynamic environments, such as e-business, demands to build adequate tools to manage multiple interactions with efficient expressiveness. This problem remains one of the most important challenges in cooperative multi-agents system research. In this paper, it is proposed an application of an efficient methodology based on Coloured Petri Nets to model multiple interactions, which takes into account the expressiveness as its most important property. The model was tested within a business to business (B2B) environment where concurrent interactions among buyers, suppliers and the marketplace constitute a dynamic process that need to be permanently monitored and controlled. This methodology provides great advantages in the representation and reasoning for the interaction mechanism modeled in cooperative information systems. The methodology integrates mainly: a) the action basic loop in order to represent the system interactions and to model organization conversations, b) the use of CPN for the interaction modeling and system simulation, c) the communicative acts of FIPA (Foundation for Intelligent Physical Agents), included in the Agent Communication Language Specification.

Keywords: Agent-Oriented Software Engineering, Cooperative Information Systems, Coloured Petri Nets, Multi-Agent Systems, Business to Business.

1. Interaction and Cooperative Information Systems

A Cooperative Information System (CIS) supports daily activities in the organization. It is a cooperative multi-agent system [24] integrated by a set of agents, data, and procedures, working in a cooperative way. They have a common goal, exchange information, and work together in order to achieve the objective [22]. The business to business systems are considered CIS and they present a set of particular characteristics, by their dynamic and interactive nature that required to be modeled in an expressive style.

To model a CIS in dynamic environments is a complex task and so the system engineer needs adequate tools in order to appropriately manage the process of modeling, in particular in this paper, the interactions among agents during the whole of information exchange. Static modeling approaches are insufficient for representing system behavior over time. The reason is that they provide no way to represent how the system's state will change as time passes. Lacking such provisions, static models can not handle dynamic interactions properly [18]; on the contrary, a dynamic modeling is one that can represent both the structure and the behavior of a system at any time of the process. In a CIS, a relevant topic is the one related with the cooperation among agents, working collectively with a

common goal, but this is not an easy modeling task, because the need to represent several collaborative layers to get a global view of the cooperative agents behavior expressively and, in particular, the interaction model of the system. In order to explain the interaction role, it is proposed to satisfy it within a more general framework, which is illustrated by the model shown in figure 1. In the model we consider that the *Cooperation* problem involves several layers: a *Communication* layer, an *Interaction* layer, and the *Coordination* layer. At the low level we have a Communication Protocol, which enables the information exchange among the agents of the system and produces a change of the system state [1], [2], [20]. The Interaction mechanism is a set of behavior rules that defines the information exchange among agents [9], [4]. The Coordination mechanism establishes the action sequence and execution according to the agents' individual goals and the common goal of the CIS [19], [8], [7]. Finally, at the high level we have a Cooperation, which is a result of the mechanisms that support interactions among the agents [17]. In figure 1, it is modeled a general cooperation situation between two agents and the relationship with the proposed framework. Here, we represent the communication layer like a link relation between agents, the interaction with a set of behavior rules, one set for each agent, and the coordination with an ordered set of actions for the different agents. It is important to point out that the order set of actions, in the coordination layer, is an order message set according to the interaction layer rules and the messages of the communication layer. By Cooperation, it is understood the agent's behavior to coordinate interaction and information exchange in order to achieve a common goal. We aim to deal more specifically with the Interaction layer in this work.

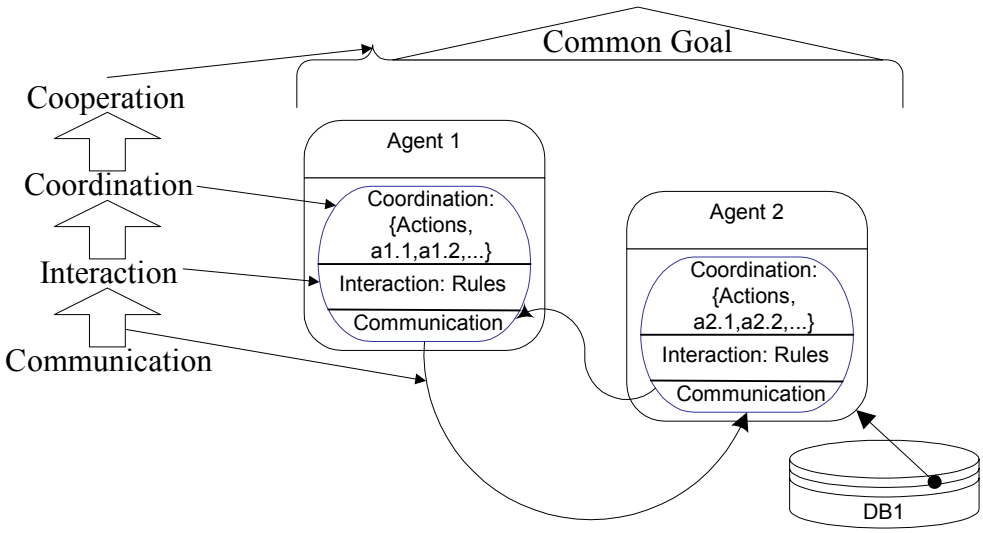


Figure 1: A Cooperation Reference Framework.

The interaction mechanism is central for the cooperation in CIS, because it is the bridge between the communication protocol and the coordination mechanism for the agents in the system. The interaction problem for CIS is immersed in a natural dynamic world, consequently, the interaction modeling and control are hard to manipulate and normally they have ambiguity and control problems. It is believed that the use of a formal method is a feasible approach to properly deal with the associated complexity in the modeling of a CIS. This will ease us to

reduce ambiguity in the interaction model and allow to simulate the dynamic of the system, which is related to multiple simultaneous interactions.

The paper is organized as follow, in section 2, it is reviewed other works in modeling the interaction using different formal methods, and therefore clarifies the most important identified problems. In section 3, an explanation is provided in how to reduce the complexity in the interaction modeling among agents using CPN, compared to other works using C/E Petri Nets. In section 4, the action basic loop is introduced to build the interaction diagrams, with the comparison between Flores's communicative acts and the FIPA communicative acts. Section 5 illustrates in detail the IMCIS (Interaction Methodology for CIS) for the interaction modeling. The methodology is explained, using an example, step by step, based on the knowledge of the action basic loop and the CPN modeling approach. Finally, the conclusion and further works are presented.

2. Interaction and Formal Methods

The uses of different formal methods in order to model the interaction mechanism are present in related literature such as: The First Order Logic [16], State Transition Diagrams [1], [14], Condition/Event (C/E) Petri Nets [12]. In the different applications of these methods, the interaction usually shows isolatedly, that is, just between two agents; however, agents are sometimes involved in several interactions simultaneously and they have to manage these multiple interactions, which involve a complex problem to be solved. Some limitations of these methods are: 1) they are practical to specify the structure of the interaction when they appear in isolated communication situations, but they are not adapted to model complex protocols with several interactions simultaneously in CIS. 2) They are very complex to manage and modify in order to respond to changes in the system specification. 3) The methods are good for representing static systems, but we need to model dynamic interactions situations, and the methods pose limitations for this. 4) The combinatory explosion of the methods when the need to model complex simultaneous interactions in CIS arise.

The most important problems to be coped with in order to improve the performance of a model oriented to these applications aforementioned are the following:

- The state of simultaneous interaction among more than two agents.
- The behavior of the agents in the interaction according to a precise state.
- The representation of different messages for different agents in different states.

The classic formal methods like the first order logic, state transitions diagrams, among others, are very difficult to manipulate, and the use of other methods such as the Petri Nets become functional, but some of them lack the expressiveness to model this kind of problem.

3. Coloured Petri Nets for Modeling Interaction

A Petri net is a formal and graphic appealing language, which is appropriate for modeling complex systems with concurrency [18]. The Coloured Petri net are high level Petri net where each token of a different color represents an arbitrary data values [21]. This extension increases the descriptive power for modeling. The firing of transitions is then made dependent on the availability of an appropriately coloured token [18]. The coloured Petri nets are a good formalism for describing concurrency, synchronization and causality [3], and are suitable for modeling, analyzing and prototyping dynamic systems with parallel activities [8], [10] as CIS in our approach. In this work, it is

proposed the use of CPN, because they have relevant characteristics for modeling interaction in CIS, such as: 1) the graphical representation, 2) the well defined semantics, 3) the formal analysis of the models and 4) the capacity for modeling the system hierarchically. The properties of CPN should allow to model the state of the interaction simultaneously among more than two agents and the behavior of the interaction according to its state, and representing different message for different agents in different states. The use of CPN computer tools, such as Design/CPN [18], helps us make a dynamic simulation of the system interaction, and to find problems before the system implementation.

Other works that model interactions using CPN are El Fallah et al. [9], [10], [11] and Cost et al. [4], [5], [6]. El Fallah et al., focuses on the study of multi-agent systems design, combining two aspects: 1) Distributed observation to capture the interactions between agents and 2) CPN as a formalism to identify interaction-oriented designs. Cost et al., focuses in the construction of a language for conversation specification, named Protolingua within the framework of the Jackal agent development environment, and they proposed use CPN as a model underlying a language for conversation specification. They do not deal with the problem of complexity and expressivities.

A central point in this work is how to reduce the associated complexity in the modeling of the interaction among agents in a CIS. In figure 2 we can observe a model used in Demazeau et al. [12] with C/E Petri Nets, where three agents or entities are represented: Entity i-1, Entity i and Entity i+1, and *m* messages, where i=1..n represent the total agents in the system, and j=1..m represent the total message number in the system. Their approach uses a "message line concept", which is modeled with C/E PN, where the virtual medium linking two agents, capable of exchanging *m* different sorts of messages, consists of *k* lines like the ones shown in figure 2, and where the dotted line represents the virtual medium. The agents are modeled with C/E PN. The C/E PN details, such as place and transitions, are evident to the system engineer, in the virtual medium, and the complexity of the model is associated with the number of links among agents and message lines. In this model the incorporation of new simultaneous interactions among agents is harder to represent, because we need to model and include new message lines, and the associated complexity is increased in m^2*s links, where *s* is the number of additional agents which are participating in the interaction and *m* is the total number of messages.

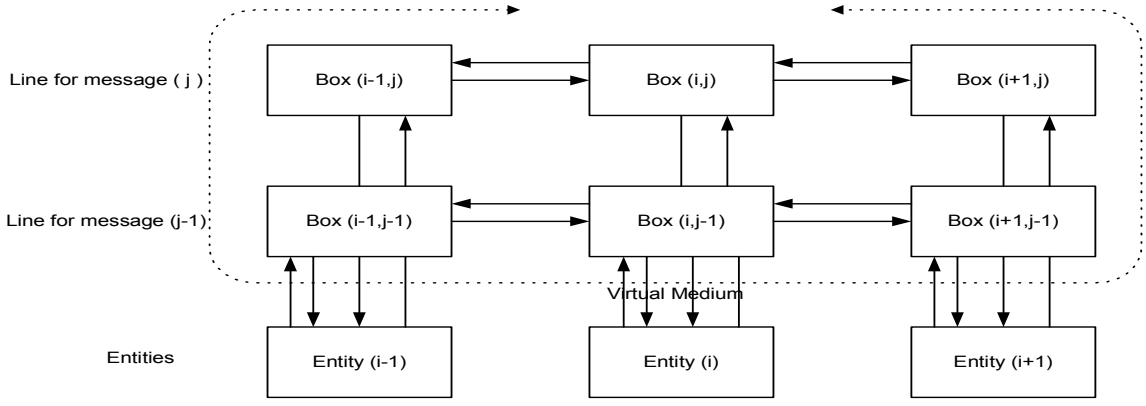


Figure 2: Demazeau Interaction Model.

It can be compared with the same problem represented in the proposed approach in figure 3, where we can see a CPN inside each agent and modeling each message that is part of the interaction relationship of the system, represented by a link between agents. CPN is not used in order to model the virtual medium, and when it is needed to incorporate new simultaneous interactions or agents, the agent and its message representation are only modified using the CPN color declarations.

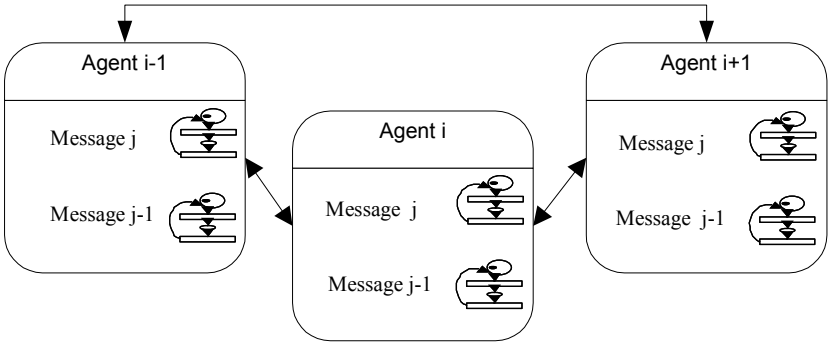


Figure 3: The Proposed Interaction Model.

If more than two agents are interacting, the use of a virtual medium to model the interaction is very difficult. Instead, by using this approach based on CPN, this difficulty is reduced considerably. We can see in [12] the associated complexity in terms of the number of links between agents interactions, according to the message numbers *(m)*, and the systems agents *(n)*. In the model presented in figure 2, the complexity is $O(m^2*n)$. While in the proposed model of figure 3, using CPN for modeling agent interaction, the associated complexity is $O(n^2)$, but, there is message independency because we just take the interactions among agents. The differences between our approach and the Demazeau approach are: 1) in the modeling of the virtual medium, such as in our approach, we use distributed systems techniques and we do not need to model this explicitly, 2) our model is independent of the number of messages among agents and 3) when modeling a new simultaneous interaction, since the associated complexity of [12] model is $O(m^2*s)$, and in our model's is $O(2n*s)$, where s is the number of additional agents which are participating in the interaction, n is the total number of agents and m is the total number of messages. Clearly, our modeling technique is suitable for a large scale agents applications, like a CIS, in contrast with Demazeau's technique, because of the latter's combinatory explosion of the method when modeling complex simultaneous interactions.

4. Interaction Modeling

Build the interaction diagrams is a central activity. Our analysis is centered on the system interactions, where we determine who talks to whom and in which way. We propose the use of the action basic loop [15] for modeling the interaction between agents. The action basic loop proposes an ontology of communicative acts: Request, Promise, Inform and Declare. In figure 4 we can see the communicative acts and their loop position, and in figure 5 we present the loop processes.

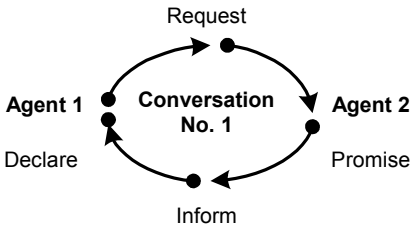


Figure 4: Communicative Acts in the Action Basic Loop.

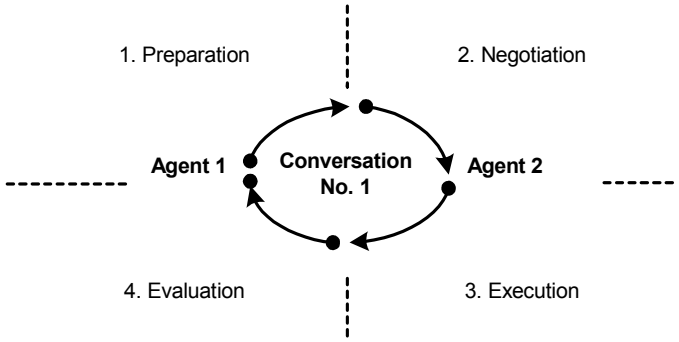


Figure 5: The Action Basic Loop.

In the first step, agent 1 *prepares* the *request* for agent 2 in a conversation. After that, agent 1 and agent 2 make a *negotiation* about the request, and agent 2 issues a *promise*. In the next stage, agent 2 *executes* the *promise* and when they finish the task, they give an *inform* to agent 1. In the last step, agent 2 makes a *declaration* for the *evaluation* of agent 1 work.

Many conversations are part of an interaction, and we need to build an interaction diagram, as the one shown in figure 6, for each system interaction. The methodology proposes a documentation set and a notation for the interaction model, but they are not shown in this paper.

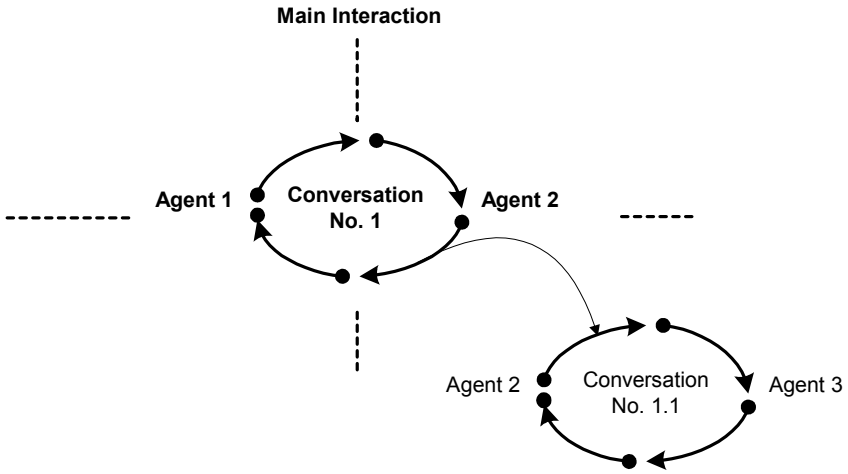


Figure 6: Interaction Diagram.

The FIPA [13] communicative acts are more expressive than the Flores's communicative acts, for instance, the FIPA ontology sets different types of *request* acts whereas Flores sets a single *request* act. It is proposed a categorization of the FIPA communicative acts according to Flores communicative acts, in table 1. Our proposal is to use the Flores's communicative acts, for the action basic loop, but exploring the FIPA approach for improving model expressiveness because, if we only use the four Flores basic acts, our communication language will lose expressivity.

The basic action loop helps us to have a coordinate conversation between agents, and to join more that two agents simultaneously in the coordinate conversation, building an Interaction. The works of El Fallah et al. [9] and Cost et al. [4], have different interaction protocols and they increase the complexity, because the agents need to recognize the interaction being used and its different states, in order to have a set of behavior rules that defines the information exchange.

FIPA	Communication for action (Flores)			
Communicative act	**Request**	**Promise**	**Inform**	**Declare**
accept-proposal				X
Agree				X
Cancel			X	
Cfp (call for proposals)	X			
Confirm			X	
Disconfirm			X	
Failure			X	
Inform			X	
Inform-if (macro act)			X	
Inform-ref (macro act)			X	
not-understood			X	
propose		X		
query-if	X			
query-ref	X			
Refuse			X	
reject-proposal				X
request	X			
request-when	X			
Request-whenever	X			
subscribe	X			

Table 1: FIPA and Flores Communicative Acts.

5. *IMCIS*: The Methodology for Modeling Interaction in CIS [23]

CIS are complex due to their dynamic nature and the management of many simultaneous interactions, and the software specification is hard to be implemented due to the reasons stated above. A proper structured way of building software for the aforementioned needs is relevant to ease the specification of complex systems. In addition to this, the problem of building powerful software tools to specify dynamic complex systems, such as CIS, has not been dealt with enough. We center our work in two areas: analysis and design of interactions in CIS. We actually hold different views about the system: 1) the explicit analysis and specification for the static view and 2) the implicit analysis and specification for the dynamic view. Current works by different authors give us a static model, but the CIS are very dynamic [22]. It is proposed to build a behavior model using the individual and the structural model. In order to make a model of a system, we need a set of abstractions that will allow us to capture the essence of the behavior of the system we wish to model. The CIS frequently perform complex tasks that are distributed over space and time, and that involve discrete flows of objects and/or information. It is proposed the use of CPN in order to represent the agent behavior and its intentions, and to simulate the resulting model. Figure 7 shows the integration of the models. At the low layer we have the *individual model*, in which we describe the agents separately. At the medium layer, we have the *structural model*, in which we describe the interactions among agents, and finally, at the upper layer, we have the *dynamic model*, in which we observe and control the simulation of the system interaction.

The explicit model is built from the system specification, but the dynamic model is built from the explicit model and simulated in the tool. The CPN model captures both the static and the dynamic behavior of the specification.

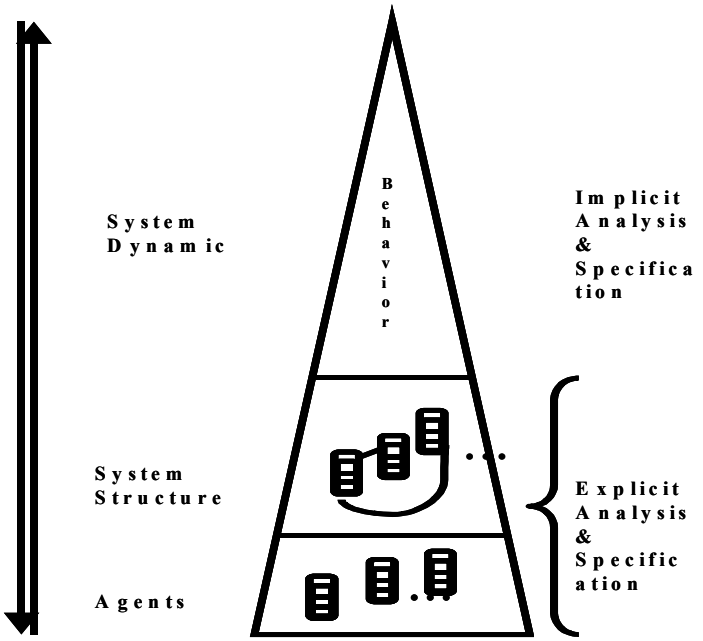


Figure 7: Methodology Views.

The IMCIS [23], the methodology for modeling interaction in CIS has the following steps:

IMCIS *(Interaction Methodology for CIS)* .

Explicit Analysis and Specification:

1. Identify agents and their intentions *(individual model)*.
2. Build the agents diagram *(structural model)*.
3. Build the interaction diagrams *(structural model)*.
4. Design the agent ports *(structural model)*.
5. Design the messages *(structural model)*.
6. Specify the systems messages using Coloured Petri Nets *(structural model)*.

Implicit Analysis and Specification:

7. Simulate the system interactions *(dynamic model)*.

5.1. Identify agents and their intentions and build the agents diagram

The methodology can be exemplified with the following case: a business to business (B2B) environment involves simultaneous interactions among buyers, suppliers and the marketplace that need to be controlled. A detail description of the methodology is shown in [23]. The first step is to identify the agents and their intentions. Here we have three agents: *Buyer*, *Marketplace* and *Supplier*. We build the agents set A, with agents and his intentions: A={(Buyer, to buy a product with the best option in the market), (Marketplace, to make effective relationships among business partners easier), (Supplier, to offer the best quality - price - volume products)}. In figure 8 we present the agent diagram, spawned from the second steep.

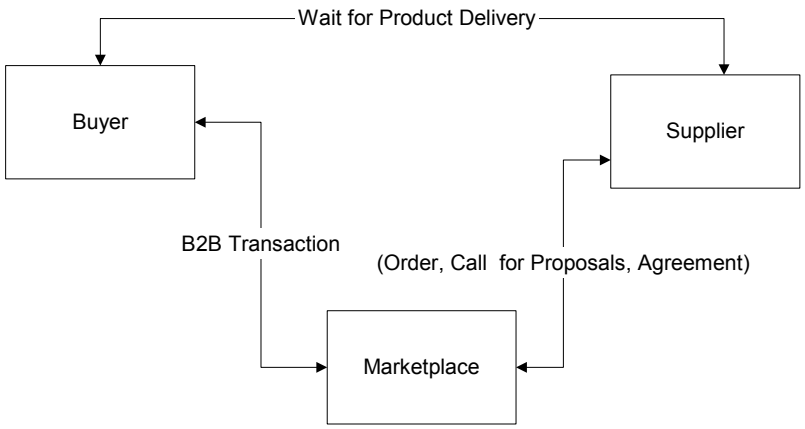


Figure 8: Agent Diagram.

The equivalent graph for the agent diagram is: AD={(Buyer, Marketplace, Supplier), (*B2B Transaction*, *Call for Proposals*, *Order*, *Wait for Product Delivery, Agreement*)}.

5.2. Build the interaction diagrams

To build the interaction diagram, the main system interaction is identified as: B2B Transaction, which is composed of four conversations: Buyer *B2B Transaction* Marketplace, Marketplace *Call for Proposals* Supplier, Marketplace *Order* Supplier, Buyer *Wait for Product Delivery* Supplier. The interaction Marketplace *Agreement* Supplier is a Context Interaction, and just gives us prerequisite information or a reference information. Figure 9 shows the interaction diagram.

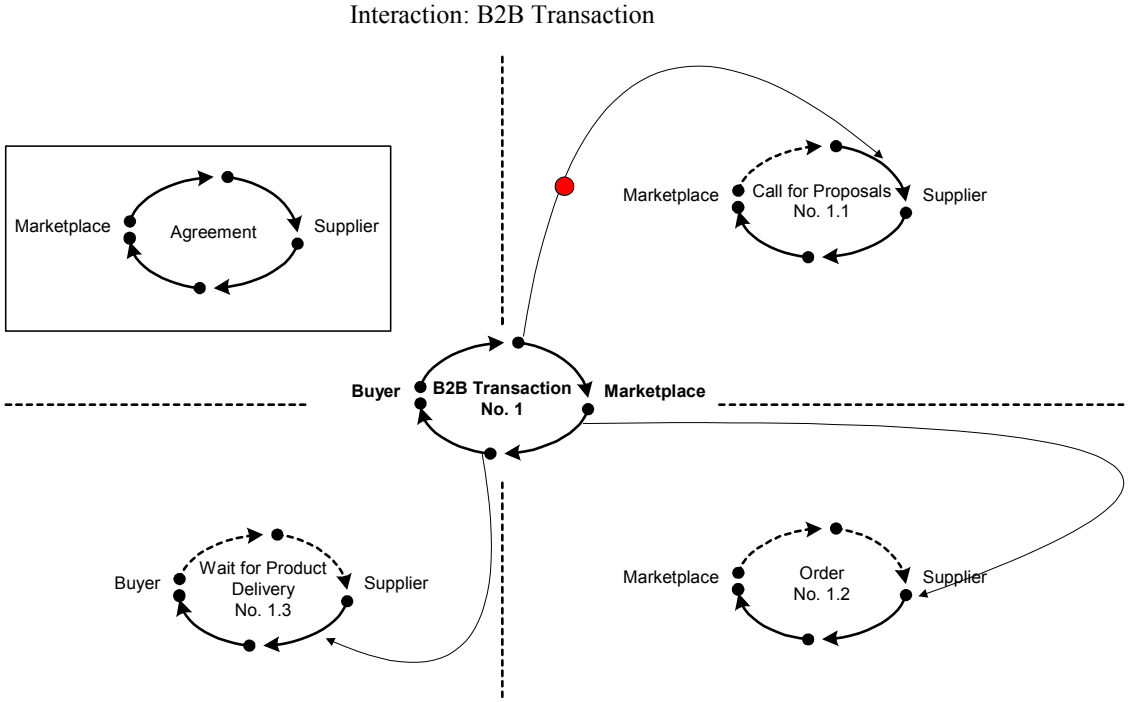


Figure 9: Interaction Diagram B2B Transaction.

So a CIS specification is represented by the following expression: CIS={A, Cg, Gk, I}, where:

- Agent Set, A={(Buyer, to buy a product with the best option in the market), (Marketplace, to make effective relationships among business partners easier), (Supplier, to offer the best quality - price - volume products)}.
- Common Goal, Cg= To satisfy the buyer need with the best offer, and the best agreements with suppliers.
- Global Knowledge, Gk={Buyer needs, products, agreements}.
- Interaction Set, I={B2B Transaction}.

In the next step, the design of the interaction in the CIS. In tables 2 and 3, the different steps and cases in a conversation are modeled, where the client/provider communicative acts [15] take place according to the steps in the basic action loop and its role in the conversation.

	Interaction			
Basic Communicative Acts	Preparation	Negotiation	Execution	Evaluation
Client				
Request	Start			
Declare satisfaction				
No agr				Start
No rep				Start
Void				Use
Cancel				
No agr		Use		
Preparation	Start			
Void				Start
Cancel make new request		Start		
Decline to accept				
No agr		Start		
Void			Start	
Close				
Revoked				Start
Declined		Use		Use
Ask recons				
Revoked		Start		
Declined		Start		
Counter		Use		
Decline counteroffer		Use		
Agree to counteroffer			Start	

Notation:
Start: Conversation Firing.
Use: Conversation Use.

Table 2: Client Communicative Acts.

	Interaction			
Basic Communicative Acts	Preparation	Negotiation	Execution	Evaluation
Provider				
Agree			Start	
Decline		Start		
Counteroffer		Start		
Revoke and counteroffer		Start		
Revoke				
No agr		Use		
Void			Use	
Report completion				
No agr				Start
Void				Start
Ask recons				
Cancel		Start		
Close				
Canceled				Use
Satisfied				Use

Notation:
Start: Conversation Firing.
Use: Conversation Use.

Table 3: Provider Communicative Acts.

5.3. Design the agents ports and the messages

The design of agents ports helps us to reduce the modeling complexity, in particular the links number among agents. The design of the agents ports is shown:

Interaction: B2B Transaction (Buyer, Marketplace, Supplier).

Conversations: 1) Buyer *B2B Transaction* Marketplace, 2) Marketplace *Call for Proposals* Supplier, 3) Marketplace *Order* Supplier, 4) Buyer *Wait for Product Delivery* Supplier.

Ports: The communicative acts are used, according to the basic interaction loop. The Buyer, Marketplace, and Supplier play two different roles in the conversation: Client and Provider. In the Pin (Port In) and Pout (Port Out) ports, the *provider* and *client* conversation messages are represent. The notation for port representation in Agent 1 Conversation Agent 2 is: Agent1.P(in or out) $_{\text{Conversation}}$, Agent2 (*m1, m2, ..., mx*), where *mi* is the *i* message in the port in the conversation between Agent 1 and Agent 2, and *x* is the total message number. The graphic representation is shown in figure 10.

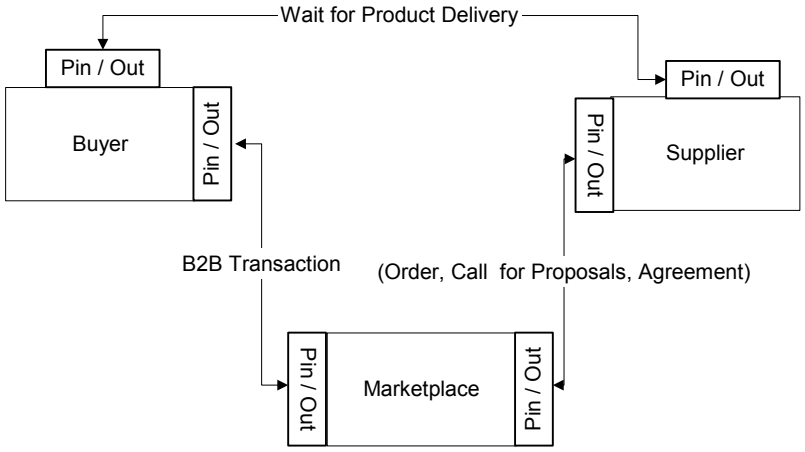


Figure 10: Agents Ports Diagram.

The *Buyer* ports are:

- Buyer.Pin $_{\text{B2B Transaction}}$, Marketplace (Agree, Decline, Counteroffer, Report Completion(Type), Revoke(Type), Revoke and Counteroffer, Close(Type), Ask Recons(Type)).
- Buyer.Pout $_{\text{B2B Transaction}}$, Marketplace (Request, Declare Satisfaction(Type), Cancel(Type), Cancel Make New Request, Decline to Accept(Type), Close(Type), Ask Recons(Type), Counter, Decline Counteroffer, Agree to Counteroffer).
- Buyer.Pin $_{\text{Wait for Product Delivery}}$, Supplier (Agree, Decline, Counteroffer, Report Completion(Type), Revoke(Type), Revoke and Counteroffer, Close(Type), Ask Recons(Type)).
- Buyer.Pout $_{\text{Wait for Product Delivery}}$, Supplier (Request, Declare Satisfaction(Type), Cancel(Type), Cancel Make New Request, Decline to Accept(Type), Close(Type), Ask Recons(Type), Counter, Decline Counteroffer, Agree to Counteroffer).

The *Marketplace* ports are:

- Marketplace.Pin $_{\text{B2B Transaction}}$, Buyer = Buyer.Pout $_{\text{B2B Transaction}}$, Marketplace.
- Marketplace.Pout $_{\text{B2B Transaction}}$, Buyer = Buyer.Pin $_{\text{B2B Transaction}}$, Marketplace.
- Marketplace.Pin $_{\text{Call for Proposals/Order/Agreement}}$, Supplier (Agree, Decline, Counteroffer, Report Completion(Type), Revoke(Type), Revoke and Counteroffer, Close(Type), Ask Recons(Type)).
- Marketplace.Pout $_{\text{Call for Proposals/Order/Agreement}}$, Supplier (Request, Declare Satisfaction(Type), Cancel(Type), Cancel Make New Request, Decline to Accept(Type), Close(Type), Ask Recons(Type), Counter, Decline Counteroffer, Agree to Counteroffer).

The *Supplier* ports are:

- Supplier.Pin $_{\text{Call for Proposals/Order/Agreement}}$, Marketplace = Marketplace.Pout $_{\text{Call for Proposals/Order/Agreement}}$, Supplier.
- Supplier.Pout $_{\text{Call for Proposals/Order/Agreement}}$, Marketplace = Marketplace.Pin $_{\text{Call for Proposals/Order/Agreement}}$, Supplier.
- Supplier.Pin $_{\text{Wait for Product Delivery}}$, Buyer = Buyer.Pout $_{\text{Wait for Product Delivery}}$, Supplier.
- Supplier.Pout $_{\text{Wait for Product Delivery}}$, Buyer = Buyer.Pin $_{\text{Wait for Product Delivery}}$, Supplier.

The Conversation Specification form shown in figures 11 and 12 is part of the documentation form set. Here the basic information about the interaction and its conversations are described. In the example, the conversation *Call for Proposals* between Marketplace and Supplier is shown. The Conversation specification form has two parts, the conversation description shown in figure 11, and the interaction diagram shown in figure 12.

Conversation Specification:

Interaction Name: *B2B Transaction*	Interaction ID: *1*	Date: *July 2001*	Version: *1.0*
Conversation Name: *Call for Proposals*		Conversation ID: *1.1*	
Main Conversation Goal: *To receive the best Supplier offers to the Marketplace according to the Buyer's request.*			
Client: *Marketplace*		Provider: *Supplier*	

Figure 11: The Conversation Description.

In the conversation diagram, the exchange of communicative acts between the Marketplace (Client) and Supplier (Provider) in each step of the action basic loop are described. All conversations in the interaction diagram must have a conversation diagram like the one shown in figure 12. The messages are represented by the communicative acts and they are present in each basic loop step, in tables 2 and 3.

In our example, the conversation *Call for Proposals* is simple, because the marketplace request product information and economic proposals, inside a previous agreement, and the negotiation space is closed to this. The suppliers can or can not send proposals, but the marketplace has the decision to accept or reject this, according to the agreement. When an agents are involved in an interaction, the conversation flow is controlled by the basic action loop. For example, in figure 12, at the *evaluation* step, if the marketplace returns the message **Declare Satisfaction**,

the conversation comes to an end, but if the message is **Decline to accept**, the conversation moves forward to the *execution* step.

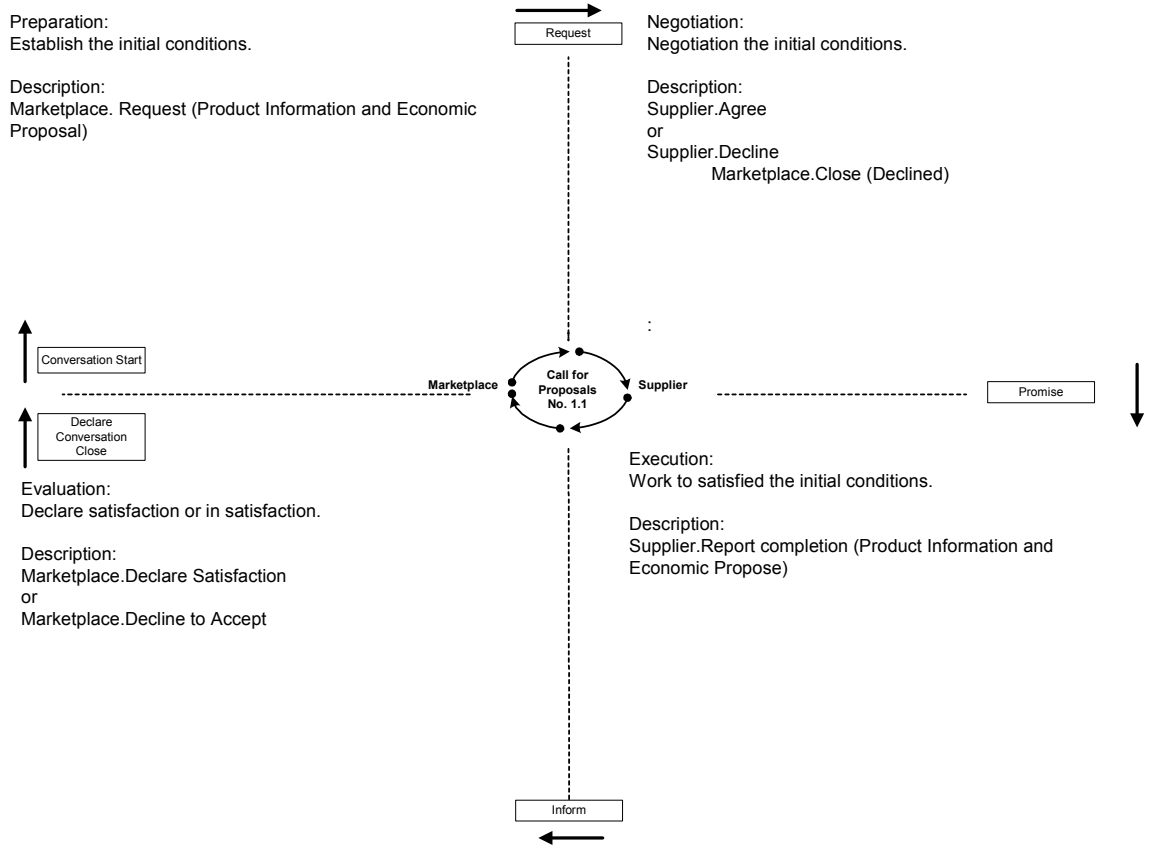


Figure 12: The Detail Interaction Diagram

5.4. Specify the System Messages Using CPN and Simulate the System Interactions

The general interaction mechanism used by the Buyer, Marketplace and Supplier is modeled. Figure 13 shows the hierarchy page *BasicActionLoopForInteraction*, build in Design/CPN, where the two agents, Client (Marketplace) and Provider (Supplier), are represented and the different pages and their relationships are shown. The Client is represented by the *MainClient* CPN, and the Provider by the *MainProvider* CPN. The *fusion place* mechanism is used for interconnecting net structure on different pages. A *fusion place* is a place that has been equated with one or more other places, so that the fused places act as a single place with a single marking [18]. The defined fusion set is *CommunicationMedium* and it represent the common places in the interaction. The Color declaration of the *Tinteraction* type is a record with a list of buyers, a marketplace, a list of suppliers, a product and a communicative act in the interaction. The buyer, supplier, marketplace and product have its own color definition. The *Tinteraction*, represent a system interaction, for example, an instance like ((Buyer 1), Marketplace, (Supplier 1, Supplier 2, … , Supplier n), Product x, Marketplace.Counteroffer)). In the following figures, *interaction* and *interactiontemp are* a *Tinteraction* declared variables.

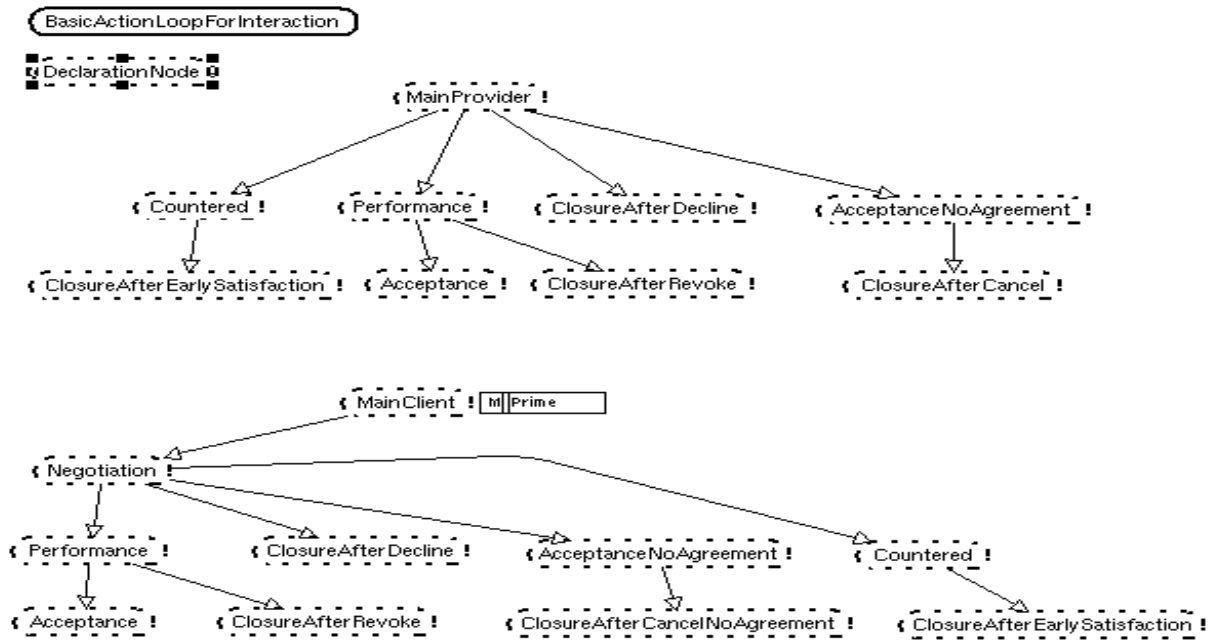


Figure 13: The General Interaction Mechanism CPN Hierarchy Page.

The first step in a general conversation shown in figure 14 is the *preparation* step in the *MainClient* CPN, and the conversation begins with a client message **request**, and the provider can respond with different kinds of messages such as **agree**, **decline**, **report completion with no agreement**, and **counteroffer**, inside the *negotiation* step.

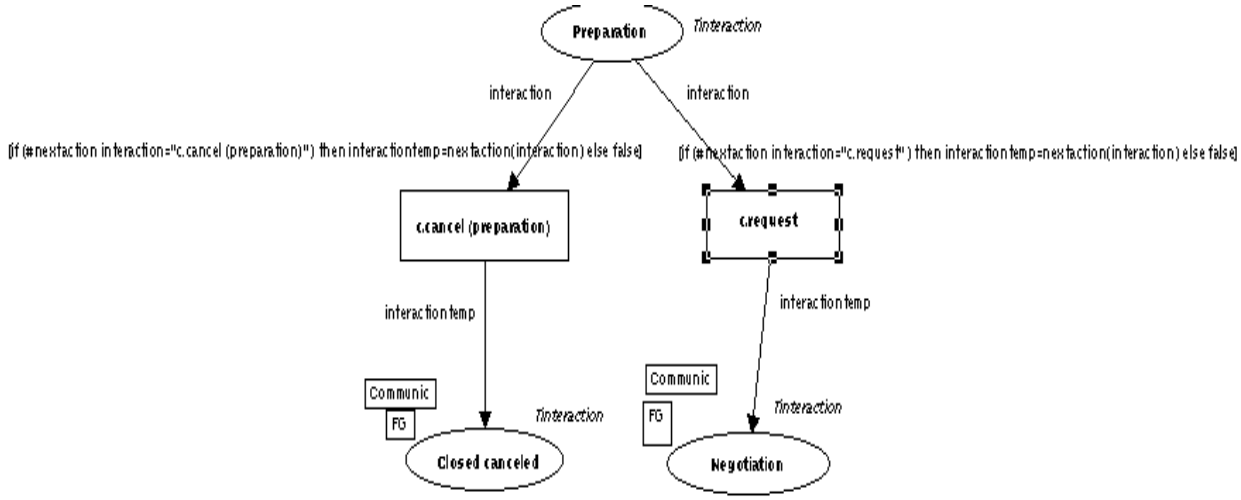


Figure 14: The Preparation Step CPN.

In a common interaction, the coordination communicative flow is:

- client **request**
- provider **agree**
- provider **report completion**
- client **declare satisfaction**

but, if the agents face an interaction conflict, a possible coordination communicative flow is: client **request** and the provider has different options:

- **Agree** to accept the requirement
- **Decline** the request
- **Report completion** with no agreement
- **Counteroffer** to provide a new option

In our example the client **counteroffer** the provider with a different product, the client has different options:

- **Counter** to ask for a different option
- **Decline** counteroffer
- **Cancel and make new request**
- **Cancel** no agreement
- **Declare** satisfaction no agreement
- **Agree** to counteroffer

If the option is **agree to counteroffer**, the interaction flow is similar to the common interaction flow. This interaction situation is part of the *negotiation* step in the *MainProvider* CPN, shown in figure 15, where the client and provider have a message exchange before advancing to the next step, as seen in tables 2 and 3. In our example, the *Call for Proposals* conversation shown in figure 12 has the following communicative coordination flow: Marketplace.Request, Supplier.Agree or Supplier.Decline, Supplier.Report Completion, Marketplace.Declare Satisfaction or Marketplace.Decline to Accept.

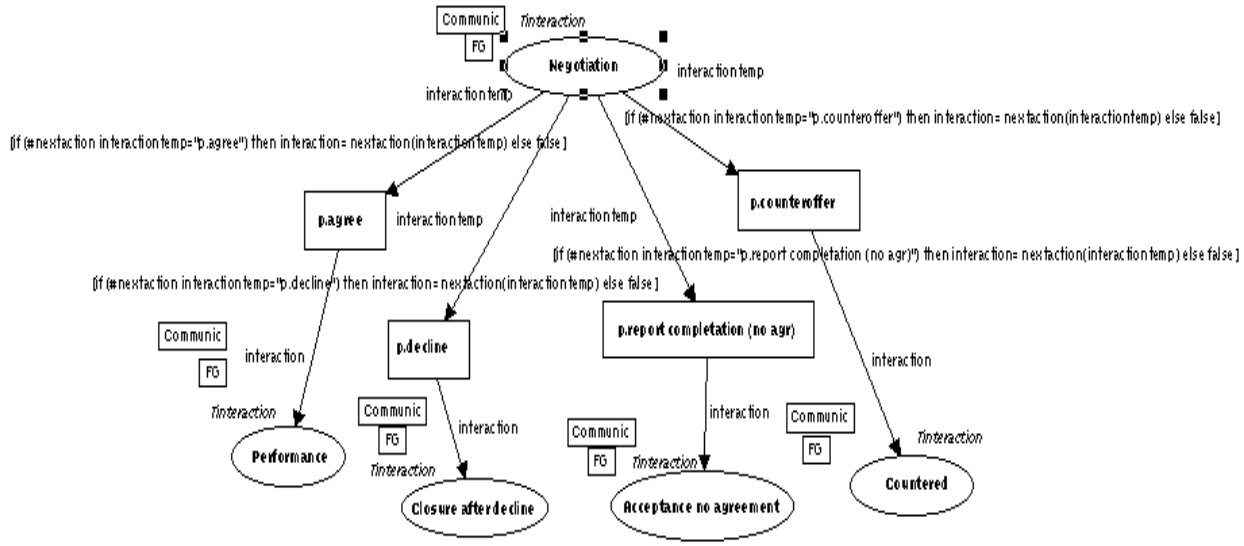


Figure 15: The Negotiation Step CPN.

Figure 16 shows part of the *negotiation step*: the c*ountered* situation, where the client or provider can **accept**, **reject** or **counteroffer** the request. The *countered* step is central in the interaction mechanism in order to resolve conflicts and make agreements.

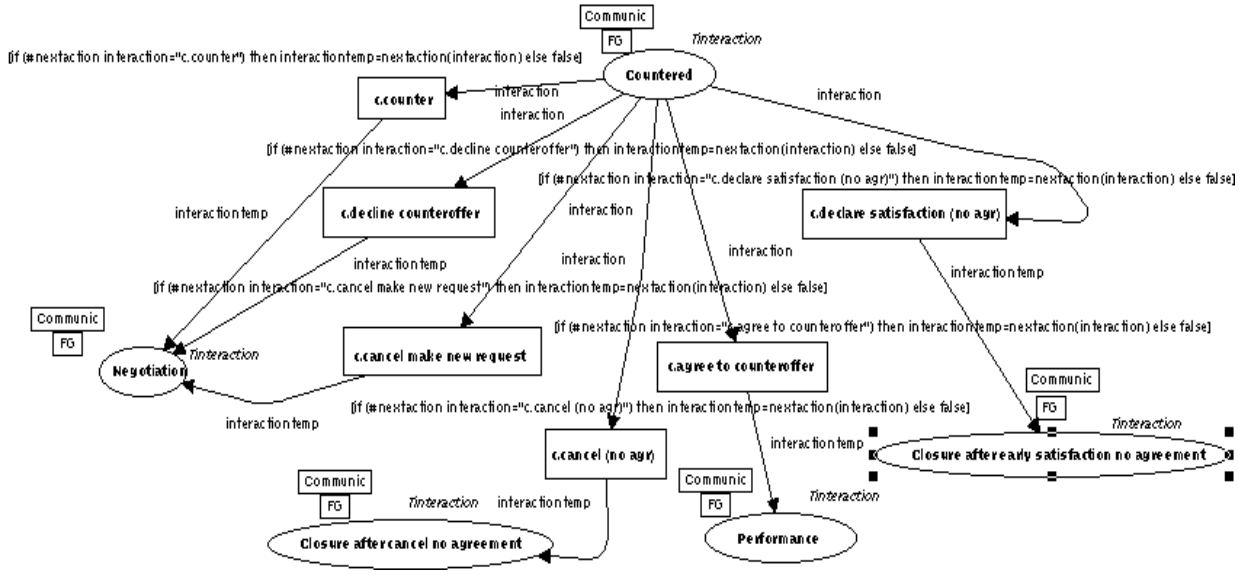


Figure 16: The Countered Step CPN.

Figure 17 shows the *execution* step, where the provider works in order to satisfy the client request. The provider can use the following messages: **report completion**, **revoke**, **revoke and counteroffer**. In the case of the **revoke** message, the conversation probably returns to the *negotiation* step, and with the **report completion** message the conversation moves forward to the *evaluation* step.

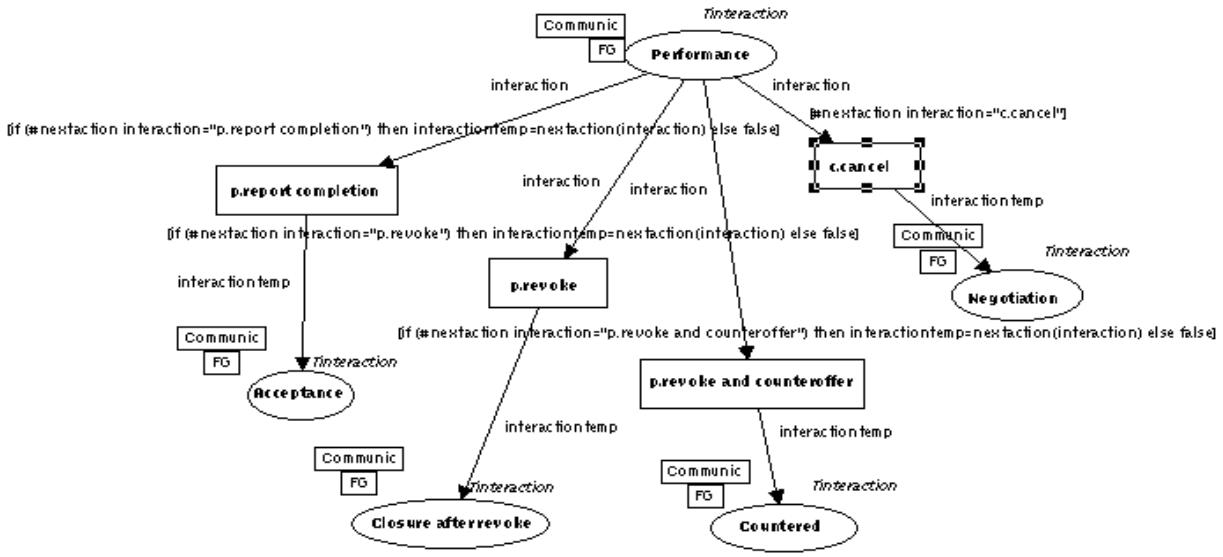


Figure 17: The Execution Step CPN.

When the conversation is at the *evaluation* step, presented in figure 18, the client must evaluate the provider's work result. The messages are: **declare satisfaction**, **decline to accept**, **cancel** or **cancel and make new request**. If the messages are **declare satisfaction** or **cancel**, the conversation comes to an end, but if the message is **decline to accept**, the conversation returns to the *execution* step or with **cancel and make new request** message, this return to the *negotiation* step.

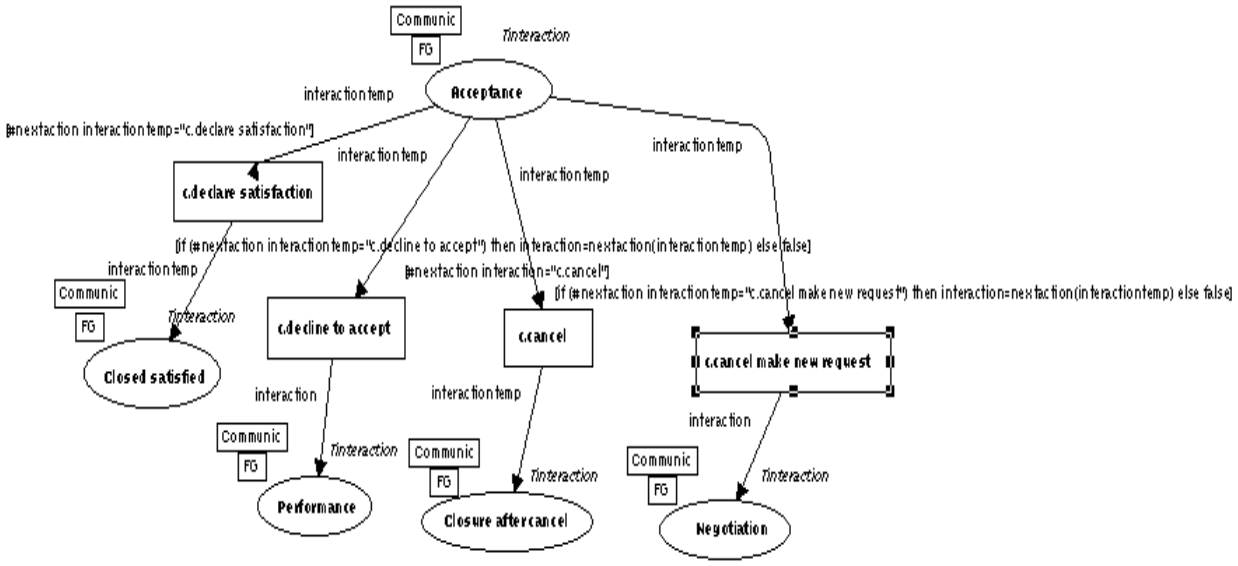


Figure 18: The Evaluation Step CPN.

An important aspect is how we model the interaction. There are two different instances of the color token interaction in the marketplace – supplier call for proposal conversation:

interaction 1 = ((Buyer 1), Marketplace, (Supplier 1, Supplier 2, … , Supplier n), Product 1, Marketplace.Declare Satisfaction))

interaction 2 = ((Buyer 2), Marketplace, (Supplier 1, Supplier 2, … , Supplier m), Product 2, Marketplace.Request))

In the instance 1, the marketplace is in an *evaluation* step within interaction 1, and in the instance 2, the marketplace is in the *preparation* step within interaction 2. Here, in the same CPN, multiple and simultaneous conversations are expressively modeled and controlled according to the token instances. Each instance has different buyers, suppliers, and products with given communicative acts within each particular conversation state.

Conclusion and Further Works

This paper provides a guide for modeling interaction in cooperative information systems by means of the use of coloured Petri nets to deal with the associated complexity for modeling the dynamic of the system. The interaction relations among agents are represented via an agent diagram, and formally specified using CPN. The use of CPN formalism offers the main advantages for modeling interaction in CIS: 1) It allows to easily model the state of the simultaneous interaction among more than two agents, 2) It allows to easily model the behavior of the interactions according to the state of the agents, 3) It allows an easy representation of different messages for different agents in different states, 4) It allows to simulate the system interaction dynamics.

The use of the basic action loop in order to model the organization interaction in the CIS helps to understand and represent different situations with a common action and coordination language, like the B2B system, but need to be tested in more application domains. The explicit and implicit analysis and specification where the system engineer may be managing the structure complexity and the system dynamics with the use of CPN. The use of *fusion place*, for modeling the system hierarchically, is a viable solution for representing the communication medium. The colors declarations helps us model complex data types and allow us to model and control multiple

simultaneous interactions among agents. An attractive area is business to consumer (B2C) systems, which are high dependent of the user decisions in the interactive model, and may required to consider temporal and fuzzy aspects that need to be modeled with the IMCIS.

References

1. P.R Cohen and H.J. Levesque, "Communicative actions for artificial agents", Proceedings of the International Conference on Multi-Agent Systems, AAAI Press, San Francisco, June, 1995.
2. P.R. Cohen and H.J. Levesque, "Rational Interaction as a Basis for Communication", Cohen P. R., Morgan, J., and Pollack, M. E. (eds.), in Intentions in Communication, SDF Benchmark Series, MIT Press, pp. 221-255, 1990.
3. J.M. Cordero, and M. Toro, "A Components Model based on Interaction-Nets", ISAS/SCI 1999, Orlando, Florida, 1999.
4. R.S. Cost, Y. Chen, T. Finin, Y. Labrou and Y. Peng, "Modeling Agent Conversations with Coloured Petri Nets", To appear in Working Notes of the Workshop on Specifying and Implementing Conversation Policies, Autonomous Agents ´99, Seattle, WA, May 1999.
5. R.S. Cost, Y. Chen, T. Finin, Y. Labrou and Y. Peng, "Using Coloured Petri Nets for Conversation Modeling ", IJCAI ´99, 1999.
6. R.S. Cost, Y. Chen, T. Finin, Y. Labrou and Y. Peng, "A Negotiation-based multi-agent system for supply chain management ", in Working Notes of the Agents ´99 Workshop on Agents for Electronic Commerce and Managing the Internet-Enable Supply Chain, Seattle, WA, April 1999.
7. K. Decker, "Environment centered analysis and design of coordination mechanisms", PhD Thesis, University of Massachusetts Amherst, 1995.
8. A. El Fallah, and S. Haddad, "A Recursive Model for Distributed Planning", ICMAS-96, p.307-314, 1996.
9. A.El Fallah, S. Haddad and H. Mazouzi, "Observation répartie et analyse des interaction dans un système multi-agents", JFIADSMA-98, Eds Hermès, Nancy 1998.
10. A. El Fallah, S. Haddad and H. Mazouzi, "Une demarche méthodologique pour l´ingénierie des protocols d´interaction", JFIADSMA-99, Eds Hermès, Nancy 1999.
11. A. El Fallah, S. Haddad and H. Mazouzi, "Protocol Engineering for Multi-agent Interaction", 9th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, MAAMAW´99, Springer, 1999.
12. Y. Demazeau, J.L. Koning, and G. Françoise, "Formalization and pre- validation for interaction protocols on multi-agent systems", Distributed AI and Multi-agent Systems, p- 298-302, 1998.
13. FIPA, Foundation for Intelligent Physical Agents, "Agent Communication Language Specification", http://www.fipa.org, 2000.
14. F. Flores, and T. Winograd, "Understanding computer and cognition, a new foundation for design", Addison Wesley, 1986.
15. F. Flores, "Introducción al Ciclo Básico de la Acción ("Loop")", Business Design Associates, Inc., 1996.
16. A. Haddadi, "Towards a Pragmatic Theory of Interactions", Morgan Kaufmann Publishers, San Francisco California, United States of America, 1998.
17. M. Huhns, and M.P. Singh, "Readings in Agents", Morgan Kaufmann Publishers, San Francisco California, United States of America, 1998.
18. K. Jensen, "Coloured Petri Nets, Basic Concepts, Analysis Methods and Practical Use", volume 1, 2, 3, second edition, Springer, Germany, 1997.
19. V. Lesser, "Reflections on the nature of multi-agent coordination and its implications for the agent architecture", Autonomous agents and multi-agent systems, Kluwer academic publishers, 1, 89-111, July 1998.
20. H. Levesque and P. Cohen, "Teamwork", Nous 25(4), Special Issue on Cognitive Science and Artificial Intelligence, pp. 487-512. To appear in: Handbook of MultiAgent Systems, 1991.
21. D. Moldt and F. Wienberg, "Multi-agent systems based on coloured Petri nets", Proceedings of the 18th International Conference on Application and Theory of Petri Nets (ICATPN´97), number 1248 in Lecture Notes in Computer Science, p-82-101, Toulouse, France, June 1997.
22. M. Papazoglou and G. Schlageter, "Cooperative Information Systems, Trends and Directions", Academic Press, San Diego, 1998.
23. F. Ramos-Quintana, J. Frausto-Solis and F. Camargo-Santacruz, "A Methodology for Modeling Interactions in Cooperative Information Systems Using Coloured Petri Nets", to appear in the International Journal of Software Engineering and Knowledge Engineering, World Scientific, http://www.ksi.edu/ijsk.html, 2001.
24. H.J. Wooldidge and N.R. Jennings, "Intelligent Agents: Theory and Practice", The Knowledge Engineering Review, 10 (2), p. 115-152, 1995.

Agent-Oriented Modelling of Distributed Systems with the Object Coordination Net Approach

Holger Giese

Software Engineering Group
University of Paderborn
hg@upb.de

Abstract. Object-oriented modelling is today the main stream approach for tackling the design of distributed systems. It permits to handle the structure and behaviour of complex real world problems. The often occurring explicit delegation between objects however results in considerable problems in the domain of distributed systems. A more flexible and open scheme for coordination is instead required. Autonomous agents which cooperate to achieve their goals rather than explicitly delegate tasks have therefore been proposed to overcome these problems. However, legacy systems supporting only the explicit delegation style have to be integrated and fine grain interaction is often better modelled using the traditional explicit delegation style. Therefore, in practice a purely agent-oriented approach is not applicable. The object coordination net approach offers an object-oriented design technique based on UML notations employing a special type of high-level Petri-Nets that permits to model distributed systems using both styles in an intermixed manner. An example is used to demonstrate how it supports the crucial aspects of distributed system design by means of standard object-oriented and agent-oriented modelling at the same time.

1 Introduction

The systematic and predictable engineering of large software systems is an inherent difficult problem. When distributed systems are considered their specific nature further results in a set of additional problems not present when designing traditional single computer systems. Roughly stating these are the phenomena of distribution, heterogeneity, coordination, resource management, and partial failures. A rather implicit resulting property is concurrency that adds its share to the complexity of handling distributed design problems. While several problems can be handled using available technologies, others remain inherent and have to be tackled in a more explicit fashion. Distribution as well as system heterogeneity are handled by applying access and location transparency offered by today's middleware systems. For the remaining aspects no comparable transparent technology mapping has been developed, which ensure that the resulting systems are predictable acceptable (cf. [35]). The great difference between network latency and computational speed makes asynchronous operating as well as parallelism in several forms mandatory when scalability or higher throughput requirements are demanded. Therefore, more autonomously operating components rather than tight coupled objects are required.

Based on the success of object-oriented programming, object-oriented analysis (OOA) and object-oriented design (OOD) [31, 36] have been developed. In contrast to structured analysis an increasingly seamless transition through the earlier stages of the software life cycle is provided. The objects and their types (classes) are more stable with respect to requirement and design changes emphasising the direct mapping concept (cf. [22]). Thus, the object-oriented paradigm is a more suitable approach for large software projects.

However, the main principles of object-orientation can be employed in quite different ways. One concept is data-driven design [31]. It starts during analysis with a domain model which is step by step extended and refined using for instance additional technical classes during the design. In contrast to data-driven design the concept of responsibility-driven design [36] tries to identify responsibilities and map them to classes rather than use classes to represent data elements of the problem domain. Therefore, it avoids both centralised and overly distributed designs. The

behaviour and data are instead well distributed and tasks are decomposed by delegation (cf. [32]). The different responsibilities of a class can be further structured when different roles [29] are distinguished. They allow to consider different responsibilities in isolation and therefore improve the separation in a design. Role and responsibility-driven design therefore results in a suitable design for distributed systems where coordinators take care of their responsibilities by coordinating other instances, while details are delegated to subordinated coordinators. This includes reactive as well as proactive behaviour. By further emphasising the coordinator role stereotype and avoiding controller stereotypes, a suitable design style for distributed systems is achieved. This design style is also appropriate for variability and adaptability, because changes will only effect other parts when the changes are so drastic that overall coordination requires adjustment.

In today's software engineering the paradigm of object-orientation is seen as the most suitable approach for the design of distributed systems (cf. [9]). The seamless support for all phases, from analysis to design and implementation and its successful concepts to handle the complexity of real software projects justifies this view. However, the common design style, which delegates behaviour to explicitly known subordinated objects, results often in a too tight coupling. Agents have been proposed for a long time [2, 34] as an alternative approach that due to the inherent autonomy of each agent avoids this drawbacks of the object-oriented approach. Today, both concepts are not further considered as contradicting paradigms. Instead, agents are rather understood as a conceptual extension of the object-oriented paradigm [27].

In agent-oriented modelling an agent is characterised by its reactive and proactive nature, its autonomy with respect to state and behaviour. An agent is further situated in some environment (cf. [38]). While compared with data-driven design the agent paradigm is quite different, a comparison with the responsibility-driven design style reveals that the essential difference is only the agent autonomy and its context awareness. Therefore agent technology can be better combined with the responsibility-driven design style which notion of responsibility does closely relate to the agent specific notion of desire.

Agents are today no longer proposed with a closed world assumption in mind. Wooldridge et al. [39] instead note that agent-based design and techniques are often more suitable employed at the coarse-grain view of a system rather than at its fine-grain view. Therefore, the agent concept can be seen rather as a specific software architectural style. The agent realization itself might be done using object-oriented technologies or special purpose agent languages. The agent-oriented paradigm reflects this by further distinguishing between a micro and macro view to denote design at different levels of granularities. The micro view describing the realization of a single agent. The macro view can be related to the coarse-grain design or software architecture.

The agent paradigm currently evolves from a programming language technology to agent-oriented software engineering [8]. Therefore, modelling techniques are required which support, besides the pure agent-oriented style, the combination with traditionally build systems. This is not only required for legacy system integration. It is also reasonable to ensure, that the agent-oriented design can use object-oriented frameworks an will be able to interact with other modules build in and object-oriented style.

The presented Object Coordination Net approach (OCoN) offers an object-oriented design technique based on UML [26] the de facto standard for object-oriented modelling. The underlying concept is a special type of high-level Petri-Nets that permits to model distributed systems supporting both the object-oriented and agent-oriented style of design. The mentioned crucial aspects for distributed system design can be modelled using the Petri net concepts contained in the visual design language OCoN. It supports both, standard object-oriented as well as agent-oriented modelling. The OCoN contract notion further supports the demanded design of coordination aspects in a modular manner. Contracts can vary from delegation to cooperation like styles, allowing a wide range of protocols. The contracts are further smoothly integrated into the language and ensure the proper coordination between its participants.

The basic concepts of the approach and a short overview is presented in the following Section 2. Then, an example is presented and several design alternatives and their systematic evaluation are discussed (see Section 3). Related work is compared in Section 4 and a final conclusion closes the article.

2 The OCoN Approach

The Object-Coordination-Net (OCoN) approach introduced in [37] provides the required integration between object-oriented and agent-oriented modelling by means of reactive as well as proactive contracts. It combines the mature UML structure diagrams with appropriate visual behaviour descriptions based on Petri-Nets [6]. The approach further seamlessly integrates the concepts of object-orientation, concurrency handling and resource coordination (see [15]). We further assume the reader to be familiar with the most important diagrams of the UML (or its predecessors) and with basic Petri-Nets [6].

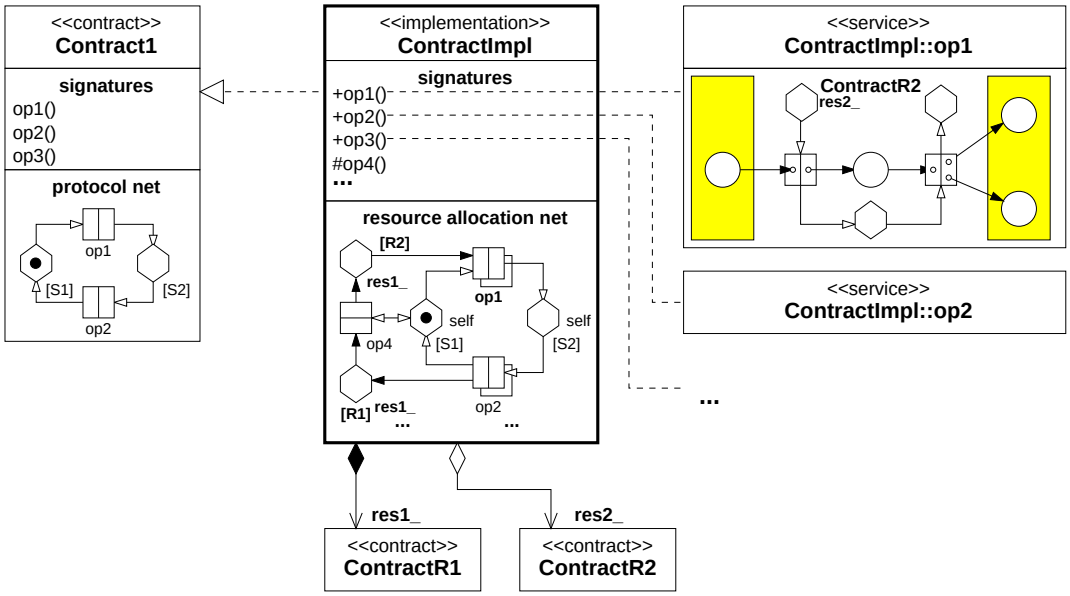


Fig. 1. UML embedding of OCoN diagrams

In order to integrate the approach with the suitable parts of the UML [26] we add only specific elements as well as new diagrams where necessary (see Figure 1). When dealing with static structure this can be done using the UML extension mechanisms. The common interface notion is extended to contracts by a so-called *protocol net* (PN) to specify nonuniform service availability. They are used to specify externally visible behaviour and to decouple systems by clearly distinguishing between an interface and its implementation. A stereotype contract is used to embed this concept into a UML class diagram (see Figure 1 contract Contract1). In class diagrams we will sometimes only use the UML interface shortcut (circle with interface name) instead of this complete stereotype. A class may provide multiple contracts to the outside which have to be fulfilled by implementing its public services. An instance-local description which resources are needed and how they are coordinated is additionally required. The UML stereotype implementation is used to denote an active UML class. Its reactive and proactive behaviour is further described by a so-called resource allocation net (RAN) as visualised for class ContractImpl in Figure 1. If a single service contains interesting internal parallelism or relevant resource allocation steps, this can by specified by a so-called *service net* (SN), otherwise a method implemented, e.g., using Java can be used which provides an interface to legacy code. A service net is analogously integrated using a service stereotype (see Figure 1 service ContractImpl::op1). Note that in contrast to hierarchical coloured Petri nets [20] and their subpage mechanism such class diagrams can describe also non-hierarchical object structures.

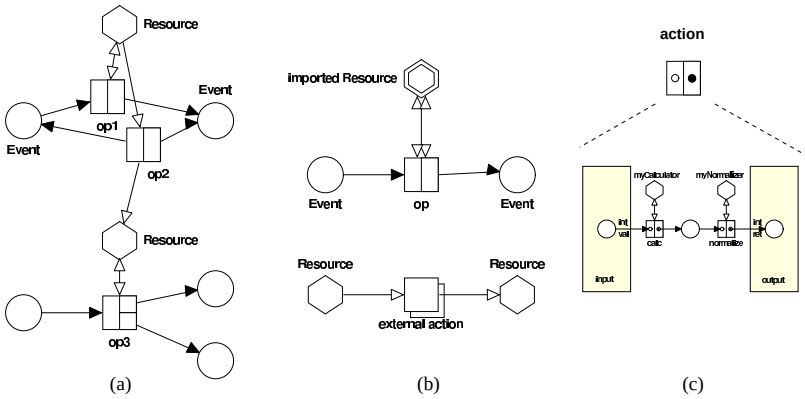


Fig. 2. OCoN elements

In Figure 2 (a) the basic elements of the additional diagrams are presented. An action is visualised by a square and represents in its basic form an operation call with request and reply part. We further distinguish two sorts of pools in form of event pools (circles) and resource pools (hexagons). The former are used to denote the control flow while the later represent the resource environment. If the resource is even not controlled exclusively within the given net a shared pool indicated by the double border is used. The flow relation of the net is described using pre- and post-condition arcs that denote which resources are consumed and produced by each action. The carrier of activity and the action executing entity is denoted by a special activation edge with white instead of black head. The actions representing an operation call are further equipped with a signature using ports that represent the parameters (white dot) as well as the replied results (black dot). If in contrast an action is drawn with a shadow and without signature it describes an externally initiated activity. This may be for example an operation call viewed from within an instance (call forward action) or an autonomous state change visible from outside (autonomous contract step).

Protocol nets describe the external visible states and operations of a contract. Resources are used to declare the different exclusive states and actions describe the operations available in each specific state of the contract. These restrictions ensure that the resulting net is a state machine.

In service nets in contrast the usage of contracts an coordination of control flow is described. Therefore the associated objects of a class may be visualised as resources. Additionally, an in and out region on the left and right side of the net (see Figure 2 (c)) are used to describe the provided parameters and required return values.

All elements of a service net with exclusion of the in and out regions may also occur in a resource allocation net to describe the request independent behaviour of a class. The resource allocation nets do further process incoming requests and therefore also contain skeleton tokens for the externally provided contracts. In the standard case of an unique provided contract case they are modelled using unnamed resource pools for each contract state. If multiple contracts with different types are provided their discriminating names are also used for the skeleton pools.

2.1 Semantics

For the purpose of the paper an informal presentation of the execution semantics is sufficient. For a more detailed description of the OCoN language and its complete semantics we refer to [13].

Figure 3 visualises how the firing of a call action is interpreted in a resource allocation or service net. The pre-condition for firing an operation like op is specified by the in-going arcs. A *carrier of activity* (here of type Res) for processing the call is required alongside the parameters (here an object of type E1) and a local pre-condition in form of a simple Event. In addition, the carrier has not only to be of type Res, but in particular must be a resource in state [S1]. This is defined by the

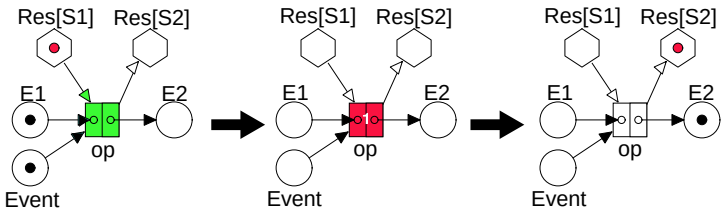


Fig. 3. The initiation and termination of a call action

type specification for the resource, i.e., Res[S1], which restricts the overall type Res to the state [S1] of Res. The firing of a call to op is composed of three elementary steps: (1) consume all pre-conditions, (2) perform some durable activity inside the service op, (3) produce all post-conditions. The effect is an object of type E2 and a change of the state of the carrier, which enters state [S2]. Therefore, an action fires essentially twice during the processing of a request, once when the request is initiated and once when such action terminates. The two steps correspond particularly well to the input and output elements of the call action. Although the direct correspondence with classical Petri net transition is abandoned, this method better preserves the integrity of an operation request which includes sending requests and receiving replies.

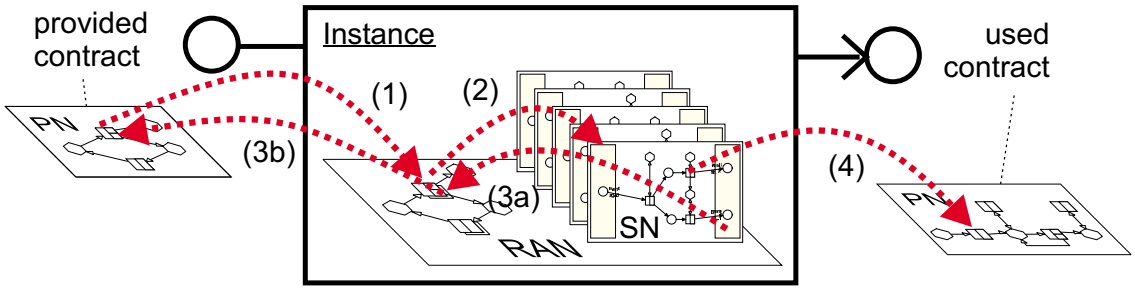


Fig. 4. OCoN architecture and net types

Besides the described local view of a call action in the calling net visualised in Figure 3, also the request processing on the server side is required. The serving object is determined by the carrier of activity. It will receive the request in its resource allocation net via a shaded action indicating that the activity is externally triggered by a client. Usually, such an action will forward the request to a corresponding service net (visualised in Figure 4). This common case of mapping a request onto a dynamically instantiated service net result in the in Figure 2(c) visualised expansion of an action into a service net. The general processing scheme for external request is as follows: (1) receive the request via a provided contract in the resource allocation net, (2) forwarded it to a dynamically created service net and (3), if required, also forward the reply back to the client. In the service nets or resource allocation net again service requests via used contracts to other object may occur (4).

This forwarding mechanism is different from the subpage mechanism of hierarchical coloured Petri nets [20] which replace each substitution node by a copy of its subpage. Besides, the more dynamic creation of service net instances the usage of the resource allocation net further results in polymorphism w.r.t. the class and state of the server object. On the client side the carrier of activity is further determined at runtime by the binding for the activation edge pre-condition (cf. [13]).

2.2 Contracts

The protocol nets, which defined the permitted usage sequences of a contract, can be used to describe strict coupling and loosely coupling contracts. The in Figure 2 (b) lower action visualised autonomous contract steps is further employed to describe proactive contract behaviour.

The contract notion does further support an efficient notion of behavioural subtyping (cf. [13]) and therefore permit to dynamically establish conform connections. The protocols can be further derived combining elements of a set of standardised protocols in such a manner that besides the protocols also full behavioural subtyping for contracts with pre- and post-conditions follows. This builds the foundation for the dynamic but secure cooperation between an evolving community of agents. Each agent collaboration is realized using matching used and provided contract types and therefore the resulting interaction excludes coordination problems. This is also true for the overall coordination when each contract is used and served independently. When an agent in contrast coordinate multiple contracts in a way that effectively synchronise their processing, overall coordination problems can in contrast not be excluded.

2.3 Contexts

In Petri nets, common resources are described by using places. By sharing such places, several system entities can exchange and share information. Place/transition nets do not distinguish different tokens located in one place and simply define enabling based upon their multiplicity. High-level Petri nets in contrast permit the specification of powerful predicates which an enabling set of tokens is required to fulfil. This ability can be used to describe associative access in the form of tuple spaces which is also a possible coordination paradigm [7].

The three basic access cases read, write and take can be easily mapped to related extended Petri net edge types. The bidirectional read edge is simply a short cut for a pre- and a post-condition edge. Whilst elegant solutions for coordination problems with this paradigm have been developed, system designs based on associative matching and tuple spaces fail however to ensure scalable system designs.

The OCoN approach supports shared pools (places) for entities of the same subsystem and thus provides a notion for the embedding of tuple spaces. Not a unstructured global tuple space but a hierarchical structured one is therefore supported. It permits the explicit import of resource pools for implementation classes and subsystems from their run-time environment (cf. [18]). Related agents can observe other ones via the contract states and appropriate activities can be started as demonstrated for the decision support agents considered later in Section 3.

The direct interaction via tuple spaces using read, write and take edges is however not type secure as connections typed via contracts. Therefore, a scheme which uses shared pools to establish connections rather than exchange data is often the more appropriate solution. This scheme realizes a local "service"-lookup using the contract type or any other additional provided meta-data offered in the shared pool. An element that has been taken or read can be used in the next step to establish the required connection in a typed manner.

3 Example

To demonstrate that traditional as well as agent-oriented object-oriented design models can be described with the approach, we will consider as example the design of an intelligent decision support system used already in [17] to discuss the visual modelling of distributed systems. It covers the collaborative management of a given set of information resources such as mail, news and channels and additionally should support client initiated data queries which results are later integrated into the information base.

Using a responsibility-driven design style, we can identify that a component is required that takes responsibility for coordinating these sets of requirements for each client. The agent [2, 34] concept can be further employed here and therefore the identified requirements can be interpreted

as desires of the autonomous operating central component. Besides the required data management and data processing, the central component has to fulfil the following complex coordination tasks: (1) each time new relevant information arrives the client wants to be informed, (2) the mail, news and channel server have to be autonomously retrieved and (3) the tasks initiate by the client have to be processed autonomously.

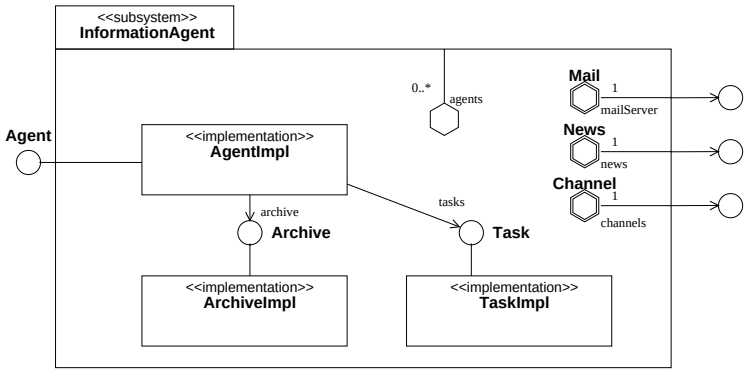


Fig. 5. The InformationAgent subsystem

In Figure 5 the relevant structural details of the agent-oriented design are presented. The offered Agent contract provides the intended functionality to a client. It is implemented by the implementation class AgentImpl using several other structural elements. Notable are at first the subsystem wide known shared resources for a mail server, a news server and a channel. They are used by the AgentImpl class to retrieve interesting information periodically. To handle specific user initiated tasks a special Task contract as well as a realization class TaskImpl are given. To store and manage the retrieved informations an ArchiveImpl class accessed via its Archive contract is used.

The presented design does however not realize the "pure" agent-oriented view. While the AgentImpl classes are conceptually agents, the style of interaction with their environment is more of an explicit delegation kind rather than the agent typical autonomous cooperation style. The mail, news and channel server contracts represent some sort of interfaces to non agent-oriented legacy system parts. In practice, we have to integrate such a system and it is not reasonable to demand a pure design with only loosely coupled contracts. Therefore, contracts can describe loosely coupling contracts as demanded by the pure agent-oriented paradigm and strict coupling ones. Each specific contract can result in a different degree of coupling and preserved autonomy for its participants.

The Mail, News and Channel contracts are traditional remote procedure based contracts which allow only interaction in the explicit delegation style. Later in Figure 9 it will be demonstrated that this style of explicit coupling requires specific treatment to preserve the autonomy of the agent w.r.t. other system components.

The TaskDesc contract illustrated in Figure 6 describes in contrast a more asynchronous interaction. The contract Task is internally used by the agent to manage a task. The initial state [working] represents that a new created task is actually processing. When the task is processed either successfully or aborted this will be represented using the state [done]. A progress autonomous contract step (white square with shadow) is used to denote that this will definitely happen in the future. Finally, an observer can obtain the result with the getResult operation when the state [done] is reached. After reading the result, the task contract does not provide any further interaction. Therefore this kind of contract is employed to manage only the life-cycle of a single task.

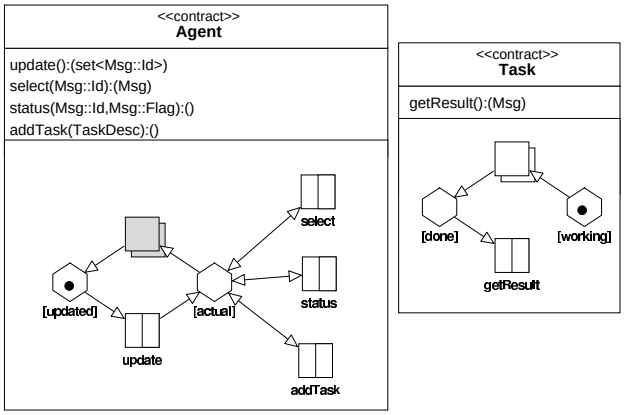


Fig. 6. The Agent and Task contracts

While the contract contains the obligation that the serving side will process the task, in contrast to the traditional contracts such as Mail, News and Channel, this time the result is provided in an additional protocol step and therefore contract client and server are more loosely coupled.

The Agent contract provided by the agent is also illustrated in Figure 6. In its state [actual] the operations select, status as well as addTask are available. Via a select request the information chunk (message) determined by a Msg::Id can be selected and obtained from the archive. A uniquely identified information chunk can be virtually modified by a specific client using the status command. If, for instance, it is marked for deletion any further interaction of that client will ignore these chunks of information. The assigned flags for an information chunk are further used to estimate their relevance and thus messages deleted by many clients may be thrown out first when the agents information base has to be reduced. If a message is created or received by a specific client it is initially only visible for the client itself. The client can either make it visible for other users or erase it using the status command.

When a new message is received or a task is completed, the contract state should autonomously change to the state [updated]. This is represented in the protocol net using a grey action with shadow indicating that the action may occur arbitrarily (quiescent autonomous contract step). Note, that in contrast to the progress autonomous behaviour this time progress is not guaranteed.

In state [updated] the client can synchronise its information base with the agent and obtain all new message id's using the update operation. When a client wants to initiate specific queries or analysis tasks, this can be done with the addTask operation using an appropriate task description (TaskDesc). The result will be added to the information base of the agent and if relevant their arrival is propagated via the [updated] state.

The contract supports a mixture of strict delegating interaction offered via the operations select, status and the autonomous behaviour in form of the [updated] state and the addTask requests. While the select and status operations are traditional remote procedure calls which deliver the intended result with the reply, the task processing is rather autonomous. The addTask request does only confirm the reception, but does not deliver the result. This is done later if any relevant results are retrieved using the autonomous state change to state [updated].

3.1 A First Design

A first solution for the AgentImpl class might be the one presented in Figure 7 which provides only a single exclusive Agent contract and does essentially realize the identified coordination. The skeleton for the provided Agent contact and its processing is described by the anonymous [actual] and [updated] pools. Two pools are used to represent the observed tasks. Two event pools and a cycling event are used to trigger the periodical retrieving of the shared mail, news and channel

server (query). The archive resource is only allocated by the different activities, but it implies no specific coordination demand.

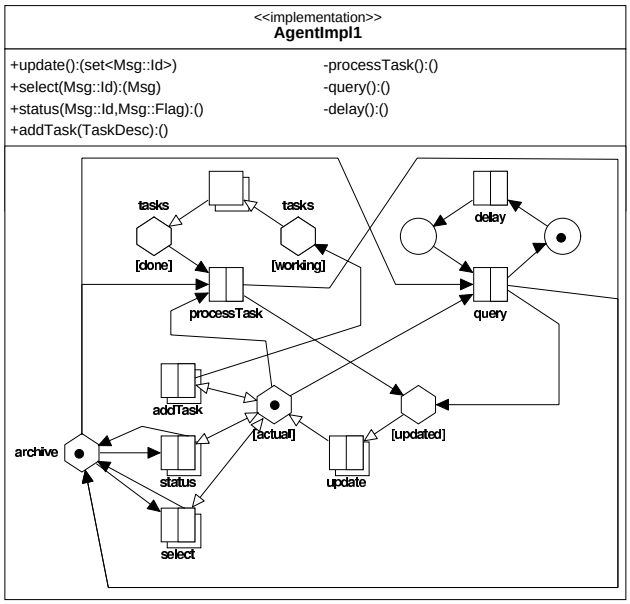


Fig. 7. A first realization AgentImpl1

The right upper part of the resource allocation net builds a control loop initially filled with a single token. The private internal operation delay is initially enabled and will terminate after a not further specified amount of time is elapsed. Then, the private method query is started which describes how the subsystem infrastructure containing a mail server, a news server and an information channel is retrieved. The archive representing the agent's information base is thus initially locked and filled with new obtained information. As additional side effect the provided Agent contract is changed from [actual] to [updated] to signal that new information chunks have been obtained to the observing client. Note, that in the external protocol the internally triggered query operation is w.r.t. its effect on the Agent contract state represented by the quiescent autonomous contract step (see Figure 6, left).

In the left upper part the initiated tasks are observed and if terminated ([done]) their result is added to the AgentImpl information base by the private processTask operation right like the query operation. Again the state of the offered contract is also adjusted accordingly.

In the lower part containing the [actual] and [updated] pool as well as the call forward actions update, addTask, status and select the offered Agent contract is realized. A skeleton token either in the [actual] or [updated] pool represents the current state of the Agent contract. The arbitrary autonomous state change depends on either the left upper or right upper part and the operations processTask or query may occur as described before. Note, that for these private operations the instance itself is the implicit carrier of activity, while for external calls the target of the request has to be specified explicitly. While the status and select operations also demand exclusive access to the archive resource does the addTask operation simply adds an additional task instance to the tasks[working] pool.

The described solution combines (a) the reactive serving of the Agent contract with (b) the polling strategy for the retrieving of the mail, news and channel server. Also (c) the observation of autonomous Task contracts is described within the same context. These three aspects and their suitable coordination are addressed using the resource allocation net.

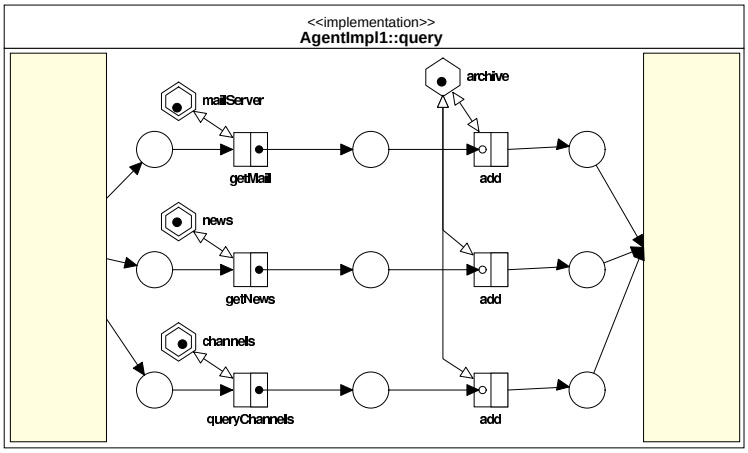


Fig. 8. The query service net

However, besides the overall coordination aspects and their interference described for an instance in its RAN, also the specific behaviour of a single operation has to be described. So called service nets as presented in Figure 8 are used for this purpose. In the AgentImpl1::query service net the strength of Petri-Nets w.r.t. explicit parallel control flow can be seen. The internally initiated query operation does further on split into three parallel threads each processing one of the mail, news and channel server. Their results are added to the overall archive exclusively locked in the resource allocation net and the operation finally terminates. Note, that the accessed context resources mailServer, news and channels are made visible by their shared resource pools.

The property that OCoN design models are executable specifications can further be used to evaluated the design. This can be used to demonstrate that the achieved degree of autonomy of the AgentImpl for the chosen design is rather limited. In Figure 9 a screen-shot of the simulation tool demonstrates that the design version of Figure 7 and 8 results in a reasonable problem w.r.t. the autonomy of the agent. Consider the case that in an agent instance one query service net is active. When additionally one of the used contracts is blocked because of a server crash or network problems the locking of the archive and provided Agent contract enforces that no external or internal operation can be processed. The blocking will remain until the crashed server or network become alive again. A scenario which is quite problematic in a distributed system.

3.2 Alternative Design

While the presented solution does ensure the intended behaviour in a consistent way via locking the archive resource wherever needed, it results in a tight coupling with the used mail, news and channel server. Therefore a rather limited degree of autonomy for the agent can be observed. If the query operation takes considerable time, the processing of client requests will be blocked until its termination. Even when no such error occurs, this might be not acceptable for an interactive client application. When mobile systems with more frequent network failures and server down-times are considered as target platform of the system, the described problem is even more problematic. An appropriate design should thus avoid this effect. One option to reduce the degree of decoupling and improve the autonomy of the agents is to further distinguishing the query operation and the archive update w.r.t. synchronisation.

By splitting the core query processing of the mail, news and channel server and adding the results to the archive this problem can be circumvented. A query operation that returns a non empty set of new retrieved messages and a processUpdate operation that takes such a set are used to decouple the processing (cf. Figure 10). We also change the processTask operation accordingly

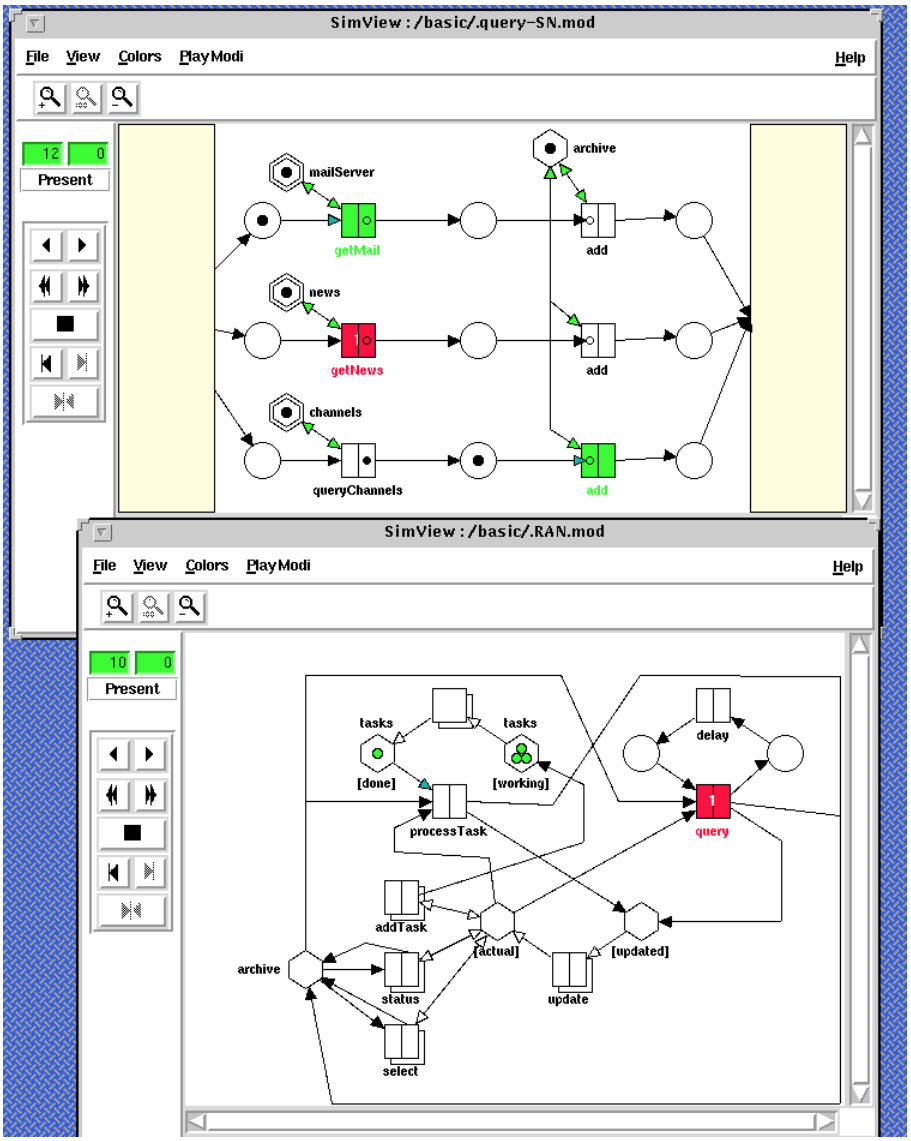


Fig. 9. Simulate a design

to avoid doubling the code describing how to add messages to the agent information base (archive). The locking of the archive resource has thus been restricted to the operations of the update part.

The query operation has also to be adjusted (see Figure 11). Instead of adding the retrieved results directly they are first united in a temporary archive (tmp) and this is returned if not empty. Otherwise no return parameter is provided.

It is to be noted that the described improvement have been achieved without changing the agent context and its contractual relations (cf. Figure 5). For a given set of provided and used contracts the achieved degree of autonomy of the agent does in contrast depend on its internal behaviour. To adjust the internal behaviour of the agent may however result in considerable semantical changes w.r.t. the degree of overall system consistency. While in the presented example the employed busy waiting strategy does result in an asynchronous update scheme anyway, in general buffering results to improve the agent autonomy does reduce the accuracy of the data. Therefore, such changes are

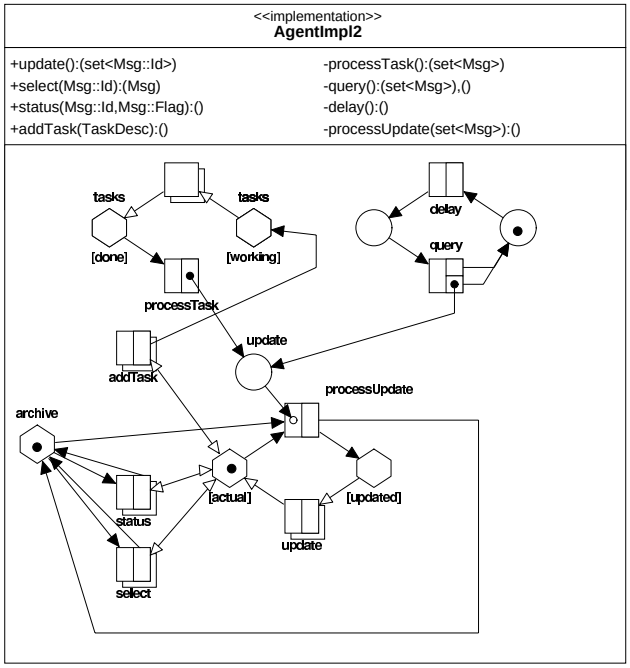


Fig. 10. Asynchronous archive update

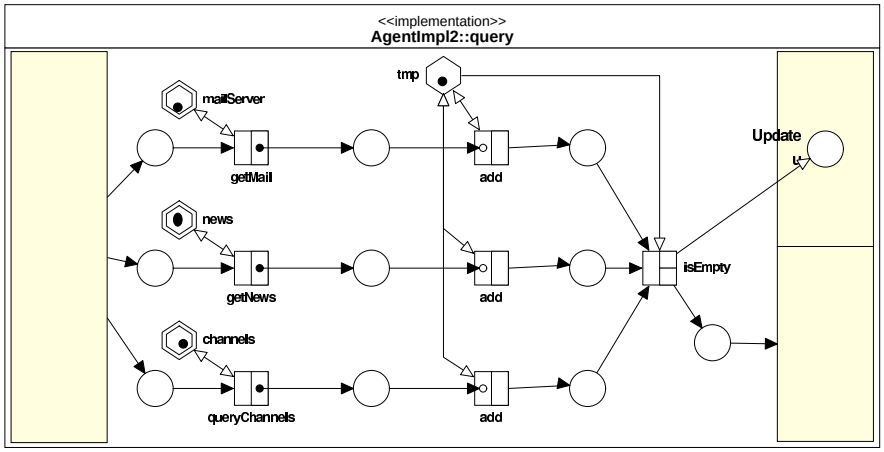


Fig. 11. The adjusted query service net

only appropriate when the overall system coordination of the redesign takes the reduce overall system consistency into account.

3.3 Multi-Client Design

If instead of a single Agent contract multiple contracts should be served by a single agent to enable a more efficient resource handling we have to further adjust the design.

Such a version serving multiple contracts at once is presented in Figure 12. The locking scheme has to be further improved and some sort of fine grain synchronisation allowing parallel access where suitable. The external locking of archive object is omitted by instead letting the archive object handle concurrent updates on its own. The specific structure of the archive can then be

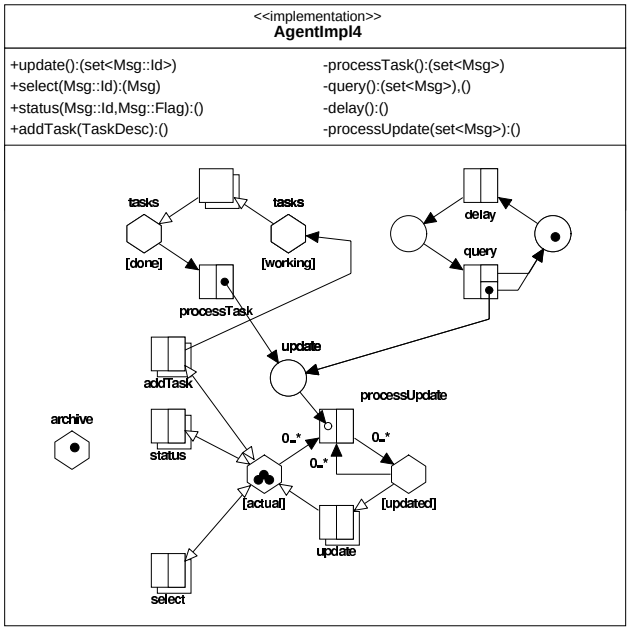


Fig. 12. A multi-contract-serving version

exploited by the object implementation to achieve parallelism where suitable and to protect the object specific consistency and semantics locally.

The resulting resource allocation net of Figure 12 is considerable simpler than earlier versions, because the resource allocation w.r.t. the archive is not needed any more. Note, however, that consistent updates are now only possible w.r.t. the granularity offered by the Archive contract. If multiple requests have to be combined to a consistent update, we have again to implement them using external locks or transaction concepts.

To serve multiple instead of a single exclusive Agent contracts also demand that instead of a single contract all of them have to be adjusted when new information is available. Instead of usual arcs, set-valued arcs described using the UML multiplicity annotation $0..*$ are used. The $*$ does denote that all existing elements are consumed rather than only the currently available. In Figure 12, all contracts in state [actual] as well as [updated] are consumed and set to state [updated] to exclude parallel processed update calls with inconsistent behaviour. Thus, pending requests may be delayed until the update is processed.

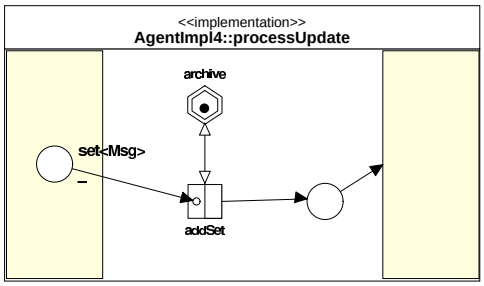


Fig. 13. Update processing with locking

The processUpdate operation realization in form of a service net has therefore to access the archive in a concurrent manner (see Figure 13). An operation addSet of the modified Archive contract is used to ensure the consistent update of the archive w.r.t. to a given set of new messages.

3.4 Use Other Agents

While the considered solutions are able to handle the requirements of a medium size configuration, we need more advanced strategies handling larger volumes of information. One limitation of the presented overall design is its very specific support for mail, news and a channel. However, in an evolving system new sources of informations may be added. Also currently useful one may later on become obsolete. Therefore, a more general solution which further exploits the agent paradigm also at the macro view is required. Instead of a single agent providing the specific service to manage mail, news, and channels, a whole web of interacting agents does better suit the described evolutionary scenario.

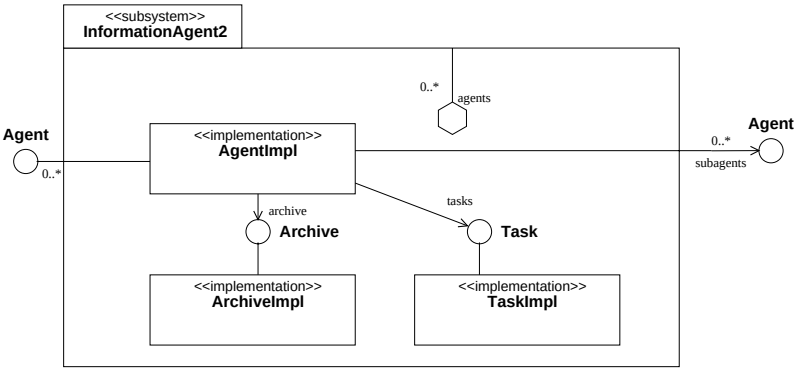


Fig. 14. The InformationAgent2

A modified version of the InformationAgent subsystem permits that also the results of agents which filters and pre-processes relevant data may be used by other agents (see Figure 14). The basic information source such as mail, news, and channels may be still integrated in the system using the beforehand described agent types. They can be used as wrappers that encapsulate the more tight coupling contracts of the mail, news and channel servers to represent them via contracts of a more agent-like cooperating fashion.

A corresponding realization AgentImpl5 for the Agent contract is shown in Figure 15. Instead of the global mail, news or channel service this time a set of Agent contracts and its update information is used to retrieve new relevant information. Thus, the right upper part contains two pools representing the different external states of the subagents contracts. They are observed and if one contract enters the [updated] state, the internal operation getUpdate is initiated. It is to be noted that thus the beforehand required busy waiting to retrieve information can be circumvented.

A layered hierarchical composition of InformationAgent and InformationAgent2 subsystems can then be used to establish the necessary infrastructure for a company wide decision support system that effectively shares common interests of several people in form of shared used information agents within the layered structure. The overall processing might be optimised by further modifying the structure, e.g., increasing the hierarchy for very busy agents.

The layered structure may be further constructed explicitly or in an flexible an adaptive manner. While the first way ensures that errorness structures such as cyclic depending agents can be excluded by construction, the resulting system has to be adjusted to changing demands manually. A far more suitable approach is to let the agents itself manage their suitable cooperation. This can be achieved when each information agent offers its services in a given context.

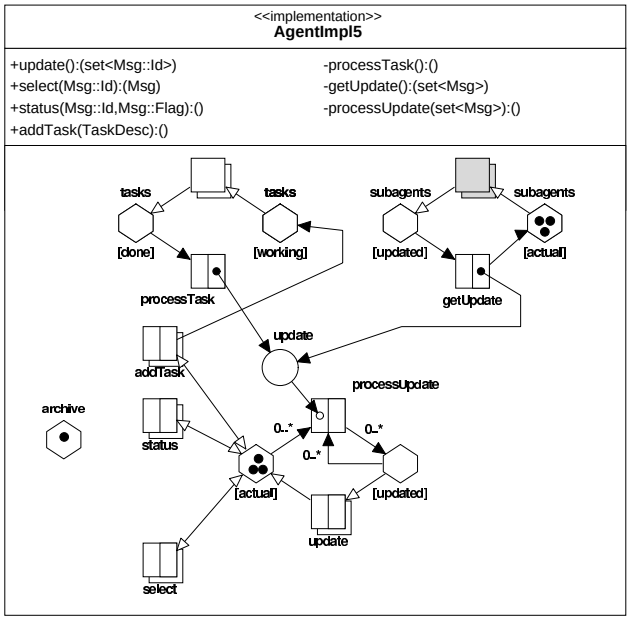


Fig. 15. The AgentImpl5 class

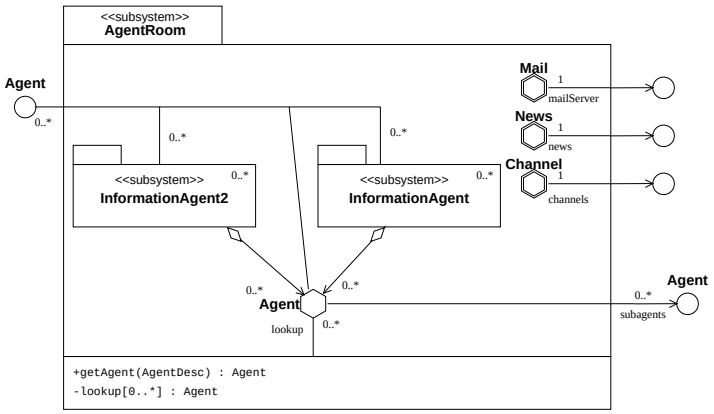


Fig. 16. Subsystem supporting dynamic cooperation of InformationAgents

In Figure 16 such a system is described using an AgentRoom subsystem which itself may contain a number of InformationAgent and InformationAgent2 subsystems. The InformationAgent subsystems share the common mail, news and channel server. Additionally, a number of InformationAgent2 subsystems using each other as well as the InformationAgent subsystems are described. The contracts are associated via shared aggregation and therefore for the resulting structural model cycles in the usage relation are excluded. The externally offered Agent contracts may be provided either by an InformationAgent or InformationAgent2 subsystem. They can be obtained from the AgentRoom subsystem interface itself.

3.5 Evaluation

The presented alternative design solutions vary w.r.t. their ensured agent autonomy and abstract performance properties to a great extend. The identified shortcomings of each design have been used to chose suitable improvement strategies to overcome the detected problems.

The OCoN approach has a well founded operational semantics as well as a mapping to the upcoming high-level Petri standard which ensures that designs can be evaluated for given usage scenarios [13]. Such a simulation may simply make the resulting synchronisation and possible effects more obvious or might result in detecting design faults. The contract protocols permit to modularise such simulations for strict hierarchical systems. For a more detailed discussion of the early evaluation of design alternatives with the OCoN approach based on UML scenarios identified during system analysis see [16].

4 Related Work

In the context of the UML there are attempts to employ the pure UML for agent-oriented modelling. Sequence and collaboration diagrams are proposed in this Agent UML dialect [28, 4] to model the overall agent interaction protocols. The importance of executable specifications has also be identified by the UML community and an approach to standardise an annotation language as well as an executable model [1] are currently developed. Describing the coordination of object-oriented and agent-oriented systems with object-oriented Petri nets permit however to overcome several limitations and problems related to behaviour modelling with the pure UML (cf. [14]).

Modularity is a rather general requirement which engineered software systems should fulfil in order to support their further evolution and change. A suitable design language requires therefore that different entities are separated by a contract notion. The pure syntactical interface notion supported by most middleware approaches is not sufficient at the semantics level. The notion of design by contract [22] has been proposed while the provided semantical aspects are pre- and post-conditions which fail to provide the necessary synchronisation information essential for distributed systems. Non uniform service availability has been identified by several researchers as an essential extension of type system w.r.t. concurrency and distributed systems (cf. [25]). Especially w.r.t. the coarse grain structure and the overall software architecture [33], higher level behavioural models and the consideration of connectors and related protocols [3] have been proposed. In contrast to the original contract notion of [22], agent contracts require a higher degree of autonomy between the contract partners and assume an agreement-oriented style, whilst programming language contracts tend to describe a form of delegation in which the serving side is strictly controlled by the client side. In distributed system design, encapsulation must be guaranteed. Thus, non type-secure approaches are not appropriate. The OCoN approach integrates an external behavioural specification. The used protocol net provides a contractual notion with behavioural subtyping and thus better support encapsulation by permitting behavioural abstraction.

Agents are a promising approach for the design of distributed systems and Petri nets have the ability to model concurrency and synchronisation. Therefore their combination has great potential to describe dynamic and flexible distributed software systems (cf. [19]). One obvious advantage is, that the analysis of system aspects become feasible [5, 30] and therefore the evaluation of the agent-based system is supported. A number of approaches for modelling of agent-oriented systems with high level Petri nets exist [23, 24, 10, 11]. They exploit the operational character of high-level Petri nets models. However, non of them provides the smooth integration with UML and a suitable behavioural contract notion that ensures correct interaction and supports dynamic contract matching.

5 Conclusion and Future Work

For the design of distributed systems approaches such as distributed objects [9] and agent-oriented software engineering [8] are current trends that will be united in the future. Therefore, the strict separation of both approaches will disappear and agent-oriented development methods such as [39] will be employed at a coarse grain design level while other parts and the design of lower granularity may be done with the object-oriented approach. Therefore, modelling techniques which support both views are required.

The presented OCoN approach supports this required smooth integration between agent-oriented and object-oriented design. Contracts ranging from an explicit delegation style towards loosely coupled cooperation can be used. However, still a well defined separation between the contract participants is ensured.

The current support for simulation of OCoN nets is not satisfactory. Further improvements w.r.t. the tool support is therefore required. As extensions for agent-oriented modelling support for agent mobility are planned in further releases of the OCoN language. Also the already existing integration with the component-based design ideas of [12], which have been not addressed in this paper, should be further tightened to achieve an integration of all three paradigms.

Acknowledgement

My colleague Ulrich Nickel contributed to this work with fruitful discussions and careful proof reading.

References

1. Action Semantics Consortium. *Response to OMG RFP ad/98-11-01 Action Semantics for the UML*, February 2001. Version 19a.
2. G. A. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.
3. R. Allen and D. Garlan. A Formal Basis for Architectural Connections. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.
4. Bernhard Bauer, Jörg P. Müller, and James Odell. Agent UML: A Formalism for Specifying Multiagent Interaction. In Paolo Ciancarini and Michael Wooldridge, editors, *Workshop on Agent-Oriented Software Engineering (Held at the 22nd International Conference on Software Engineering (ISCE2000))*, pages 91–103. Springer Verlag, 2001.
5. J. Billington, B. B. Du, and M. Farrington. Modelling and Analysis of Multi-Agent Communication Protocols using CP-nets. In *Proc. 3rd Biennial Engineering Mathematics and Applications Conference (EMAC'98), Adelaide, Australia, 13-16 July 1998*, pages 119–122, July 1998.
6. W. Brauer, W. Reisig, and G. Rozenberg, editors. *Petri Nets: Central Models (part I)/Applications (part II)*, volume 254/255 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin, 1987.
7. N. Carriero and D. Gelernter. *How to write parallel Programs, A First Course.* MIT Press, Cambridge, MA, April 1990.
8. P. Ciancarini and M. Wooldridge, editors. *Agent-Oriented Software Engineering · First International Workshop, AOSE 2000 Limerick, Ireland, June 10, 2000 Revised Papers*, volume 1957 of *Lecture Notes in Computer Science*. Springer Verlag, 2001.
9. Wolfgang Emmerich. *Engineering Distributed Objects.* John Wiley & Sons, Inc., March 2000.
10. J. M. Fernandes and O. Belo. Modeling multi-agent system activities through colored Petri nets: an industrial production system case study. In *Proc. 16th IASTED Int. Conf. on Applied Informatics, 23-25 February 1998, Anaheim, CA*, pages 17–20, 1998.
11. H. Fiorino and C. Tessier. Agent cooperation: a Petri net based model. In *Proc. 3-rd Int. Conference on Multi-Agent Systems (ICMAS'98), 3-7 July 1998, Paris, France*, pages 425–426, 1998.
12. Holger Giese. Contract-based Component System Design. In Jr. Ralph H. Sprague, editor, *Thirty-Third Annual Hawaii International Conference on System Sciences (HICSS-33), Maui, Hawaii, USA.* IEEE Press, January 2000.
13. Holger Giese. *Object-Oriented Design and Architecture of Distributed Systems.* Berichte aus der Informatik. Shaker Verlag, March 2001.
14. Holger Giese, Jörg Graf, and Guido Wirtz. Closing the Gap Between Object-Oriented Modeling of Structure and Behavior. In Robert France and Bernhard Rumpe, editors, *UML'99 - The Second International Conference on The Unified Modeling Language Fort Collins, Colorado, USA*, volume 1723 of *Lecture Notes in Computer Science*, pages 534–549. Springer Verlag, October 1999.
15. Holger Giese, Jörg Graf, and Guido Wirtz. Seamless Visual Object-Oriented Behavior Modeling for Distributed Software Systems. In *IEEE Symposium On Visual Languages, Tokyo, Japan.* IEEE Press, September 1999.
16. Holger Giese and Guido Wirtz. Early Evaluation of Design Options for Distributed Systems. In *Int. Symposium on Software Engineering for Parallel and Distributed Systems (PDSE'2000), Limerick, Ireland.* IEEE Press, June 2000.

17. Holger Giese and Guido Wirtz. Visual Modeling of Object-oriented Distributed Systems. *Journal of Visual Languages and Computing*, 12(2):183–202, April 2001.
18. T. Holvoet and T. Kielmann. Behavior specification of active objects in open generative communication environment. In *Proc. 30th Annual Hawaii Int. Conf. on System Sciences;: Software Technology and Architecture, 7-10 January 1997, Wailea, HI*, volume 1, pages 349–358, January 1997.
19. Tom Holvoet. Agents and Petri Nets. *Petri Net Newsletter*, (49):3–8, October 1995.
20. K. Jensen. *Coloured Petri Nets Basic Concepts Analysis Methods and Practical Use Volume 1*. EATCS Monographs on Theoretical Computer Science. Springer Verlag, 1992.
21. Pekka Kähkipuro. UML Based Performance Modeling Frameworks for Object-Oriented Distributed Systems. In Robert France and Bernhard Rumpe, editors, *UML'99 - The Second International Conference on The Unified Modeling Language Fort Collins, Colorado, USA*, volume 1723 of *Lecture Notes in Computer Science*, pages 356–371, October 1999.
22. Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997. 2nd edition.
23. T. Miyamoto and S. Kumagai. A Multi Agent Net Model of Autonomous Distributed Systems. In *Proc. of Computational Engineering in Systems Applications'96 (CESE'96) Symposium on Discrete Events and Manufacturing Systems, 9-12 July 1996, Lille, France*, pages 619–623, July 1996.
24. Daniel Moldt and Frank Wienberg. Multi-Agent-Systems Based on Coloured Petri Nets. In Azéma, P. and Balbo, G., editors, *Lecture Notes in Computer Science: 18th International Conference on Application and Theory of Petri Nets, Toulouse, France, June 1997*, volume 1248, pages 82–101, Berlin, Germany, June 1997. Springer Verlag.
25. Oscar Nierstrasz. Regular Types for active Objects. In *Proceedings OOPSLA'93*, volume 28 of *ACM SIGPLAN Notices*, pages 1–15, October 1993.
26. Object Management Group. *OMG Unified Modeling Language Specification, Version 1.3*, June 1999. OMG document ad/99-06-08.
27. James Odell. Objects and Agents: Is There Room for Both? *Distributed Computing*, November 1999. (www.DistributedComputing.com).
28. James Odell, H. Van Dyke Parunak, and Bernhard Bauer. Extending UML for Agents. In Gerd Wagner, Yves Lesperance, and Eric Yu, editors, *Agent-Oriented Information Systems Workshop at the 17th National conference on Artificial Intelligence (AAAI2000), Austin, TX, USA*, pages 3–17, 2000.
29. Trygve Reenskaug, Per Wold, and Odd Arild Lehene. *Working with Objects: The OOram Software Engineering Method*. Addison-Wesley/Manning, 1996.
30. P. Rongier and A. Liegeois. Analysis and prediction of the behavior of one class of multiple foraging robots with the help of stochastic Petri nets. In *Proc. IEEE Int. Conf. on Systems, Man, and Cybernetics (SMC'99), 12-15 October 1999, Tokyo, Japan*, volume 5, pages 143–148, 1999.
31. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.
32. Rovert. C. Sharble and Samuel S. Cohen. The Object-Oriented Brewery: A Comparison of Two Object-Oriented Development Methods. *ACM SIGSOFT Software Engineering Notes*, 18(2):60–73, 1993.
33. Mary Shaw and Davis Garlan. *Software Architecture: Perspectives on an emerging Discipline*. Prentice Hall, 1996.
34. Y. Shoham. Agent-oriented programming. *Artifical Intelligence*, 60:51–92, 1992.
35. Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendal. A Note on Distributed Computing. Techreport, Sun Microsystems Laboratories, November 1994. TR-94-29.
36. R. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software*. Prentice Hall, 1990.
37. Guido Wirtz, Jörg Graf, and Holger Giese. Ruling the Behavior of Distributed Software Components. In H. R. Arabnia, editor, *Proc. Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'97), Las Vegas, Nevada*, July 1997.
38. Michael Wooldridge and Nicholas R. Jennings. Agent Theories, Architectures, and Languages: a Survey. In Michael Wooldridge and Nicholas R. Jennings, editors, *Intelligent Agents*, pages 1–22. Springer Verlag, 1995.
39. Michael Wooldridge, Nicholas R. Jennings, and David Kinny. A methodology for agent-oriented analysis and design. In *Proceedings of the third annual conference on Autonomous Agents May 1 - 5, 1999, Seattle, WA USA*, pages 69–76, 1999.

An Experiment with Coordinated Algebraic Petri Nets as Formalism for Modeling Mobile Agents

Jarle G. Hulaas
University of Geneva
1211 Genève 4, Switzerland
Jarle.Hulaas@unige.ch

Didier Buchs
Swiss Federal Institute of Technology
1015 Lausanne, Switzerland
Didier.Buchs@epfl.ch

Abstract. This paper shows the aspects of CO-OPN/2, a formal component-oriented modeling language based on algebraic Petri nets, that can be used for the formal modelling of distributed applications involving mobile agent technology. The CO-OPN/2 language offers several structuring tools to make consequently sized projects manageable, and the development environment provides a rich toolset designed to support several phases of the software process, such as simulation, test set generation and production of executable code.

1. Introduction

Distributed applications designed following classical technologies, like remote method calls, have benefited from many years of research in formal methods. On the other hand, systems incorporating mobile agents exhibit new characteristics which cannot thoroughly be expressed and studied with previously developed formalisms. The difficulties stem either directly from the very concept of mobile agent (process mobility, new communication patterns, dynamic discovery of services), or indirectly from pre-existing problems which have become unavoidable with the mobile agent paradigm (security, resource management, loose coupling of software components).

This paper describes an approach relying on CO-OPN/2 (Concurrent Object-Oriented Petri Nets) [3][5], a formal component-oriented modeling language based on algebraic Petri nets. It has an associated coordination formalism COIL [6] as well as its comprehensive tool-set, for the formal development of distributed applications, including, but not restricted to mobile agent technology. The originality of our contribution is two-fold. First, it lies in the use of a Petri nets extension for the formal description of a form of object mobility which is very close to the real world, by opposition to a simulated mobility obtained by updating communication links. Second, we support different degrees of abstraction in the modeling of mobile agent applications, ranging from simple sketches to concrete, implementation-ready specifications designed with all habitual structuring principles. The benefits a software engineer can obtain from our approach are: the safety of a modeling language with well-defined semantics, a strong integration in the development process, as well as a resulting specification with a clear graphical presentation and a structure and semantics close to current implementation paradigms.

This paper is structured as follows. Section 2 introduces our formalism, and Section 3 demonstrates its usage in the case of Milner's mobile telephone example. Section 4 discusses our requirements for a mobile agent modelling language. Section 5 shows a small mobile agent example in CO-OPN/2. Section 6 presents current and future activities in this framework, and Section 7 concludes the paper.

2. CO-OPN/2 and COIL

CO-OPN/2 [3] is a specification language which permits an abstract description of concurrent operations and data structures of computer programs. CO-OPN/2 models can be developed in a dedicated environment described in [8]. This approach, as illustrated in Figure 1, proposes a formal and incremental development method which covers:

- Modeling by the means of a concurrent object-oriented specification language supporting refinement, analysis and simulation within an inference-based environment.

- Automatic generation of distributed executables with a prototyping tool, as well as incremental incorporation of hand-written code portions in order to obtain a satisfying end-user implementation [7]. A new Java-generating version of the tool is presented in [10].
- Specification-based test generation assorted with a test reduction technique founded on the introduction of hypotheses about the program [2].

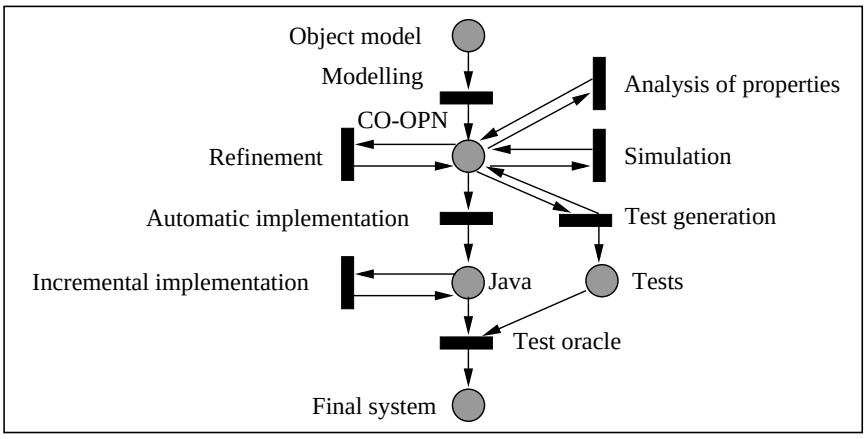


Figure 1. Activities covered in the CO-OPN/2 development methodology

Different semantics of CO-OPN/2 are exploited during these phases: an abstract operational semantics is used for the logic programming simulator as well as for the test set generation tool, while a more concrete operational semantics based e.g. on transactions for prototyping of tightly coupled distributed systems. All these semantics are valid and partially complete on a subpart of the CO-OPN/2 language with respect to its denotational semantics.

CO-OPN/2 is an object-oriented modeling language, based on ADT (Algebraic Data Types), Petri nets, and IWIM (Idealized Workers, Idealized Managers) coordination models [6]. Hence, CO-OPN/2 concrete specifications are collections of *ADT*, *class* and *coordination* modules [3][5]. Syntactically, each module has the same overall structure; it includes an interface section defining all elements accessible from the outside, and a body section including the local aspects private to the module. Moreover, class and context modules have convenient graphical representations, putting in evidence their basic Petri net model. Explaining structuring tools such as object-orientation, genericity, sub-classing and sub-typing, are out of the scope of this paper, and can be found in [3]. Similar models dealing with object references in Petri nets are [14] and [15].

2.1. ADT Modules

CO-OPN/2 ADT modules define data types by means of algebraic specifications. Each module describes one or more sorts (i.e. names of data types), along with generators and operations on these sorts. The exact definition of the operations is given in the body of the module, by means of equational axioms. For instance, Figure 2 describes a very simple ADT defining one sort (the booleans) and one operation on this sort (the negation).

```
ADT SimpleBooleans;
Interface
   Sort boolean;
   Generators true, false : -> boolean;
   Operation not _ : boolean-> boolean;
Body
   Axioms
      not (true) = false;
      not (false) = true;
End SimpleBooleans;
```

Figure 2. ADT SimpleBooleans

2.2. Class Modules

CO-OPN/2 classes are described by means of modular algebraic Petri nets with particular, parameterized external transitions, the methods of the class. The behavior of transitions are defined by so-called behavioural axioms, corresponding to the axioms in ADT specifications. Method calls are achieved by synchronizing external transitions, according to a transition fusion technique. Several synchronization operators (the `//`, `..` and `+` operators, which repectively represent the parallel, sequential and alternative composition) are provided by the language for expressing complex synchronization constraints. These constraints must be satisfied in order to fire the whole event. A parallel constraint means that the related events must occur simultaneously and consequently that enough resources are available to each event. A sequence means that the firings can occur sequentially. An alternative means that at least one of the events can fire.

Below (Figure 3) is the code and the associated Petri net graph of a class modeling a peculiar storage system; it stores boolean values, but delivers the negated ones. The interface defines two methods, for the injection and the ejection of values. The body is actually a textual representation of the associated Petri net. The flow relation has the following structure:

[*cond* =>] *event* [**With** *sync*] :: *Pre* -> *Post*;

Where *cond* is a conjunction of equalities, *sync* is a synchronization expression and *Pre* and *Post* conditions are the classical input ressources and output ressources of places in Petri nets. Free variables may be defined and used in the behavioral axioms. In their graphical representation, transitions are thin black boxes and methods black ones.

```
Class NegatingStorageSystem;
Interface
   Use SimpleBooleans;
   Type negatingstoragesystem;
   Methods put _ ,  get _ : boolean;
Body
   Place container _ : boolean;
   Axioms
      put b :: -> container b;
      get b :: container not(b) -> ;
   Where b : boolean;
End NegatingStorageSystem;
```

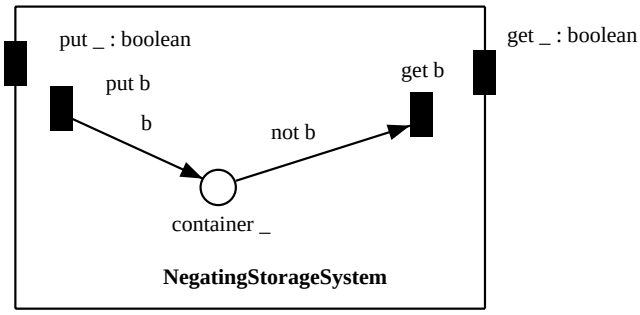


Figure 3. Class NegatingStorageSystem

2.3. COIL Coordination Modules

A third kind of module is present in CO-OPN/2, the COIL context modules [6], which exhibit the same overall structure as ADT and class modules. Basically, context modules allow modeling loosely coupled distributed systems, by means of suitable coordination mechanisms, more complex than the fusion of transitions seen above. In addition to the standard *input ports* (the methods), COIL also has a notion of *output port* (called *gate*), and both event concepts are used to enable the coordination of different components. Coordination thus basically consists of events to events connections; the same synchronization operators as mentioned above may be used to this end.

Whereas classes are the means to model tightly coupled entities, contexts provide the coarser granularity needed for modeling software components.

3. Milner's Mobile Telephone in CO-OPN/2

As an introduction to how mobile entities are modelled with contextual coordination, we demonstrate a CO-OPN/2 version of Milner's "mobile telephone" example [13]. Figure 4 recalls the flow-graph of this example. A center is in permanent contact with two bases (*base1*, *base2*) by means of two channels *gives* and *alert*. With the *gives* channel it informs the active base about

the next base for a given car, while it tells the receiving base about the arriving car using the *alert* channel. Only one car is modeled in this simple example and is initially in contact with *base1*. Both channel *talk* and *switch* are used for communication between the car and the base. The car uses the *talk* method to chat through its current base (which should further route the conversation appropriately, but this is not relevant here), which sends to the car the name of the next nearest base by means of the *switch* channel.

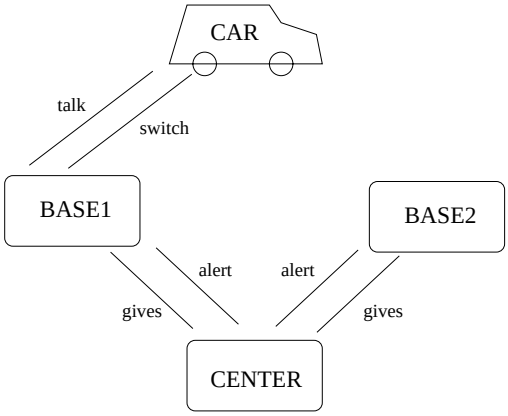


Figure 4. Flow-graph of the Mobile Telephone Example

The system is modeled by a context hierarchy: a *center* context containing two sub-contexts *base1* and *base2*. These sub-contexts are connected together, and allow objects of class *Car* to migrate between them. Both sub-contexts contain a static object *base* (of class *Base*) which manages the migration of the car. In Figure 5, contexts are represented as rounded boxes, statically instantiated objects as ellipses and dynamically created objects as square boxes. Contexts and objects have input ports (or methods) on their membrane (represented as filled black boxes). Arcs between them represent standard method calls. In addition, special output ports (or gates, represented as white boxes) are present on the membranes. Synchronizations can be more complex than just method calls, and in this case they are represented graphically by binary tree structures (Figure 6) where the nodes are circles surrounding the synchronization operator (for the non-commutative sequence operator, the orientation of the triangle gives the needed additional ordering information).

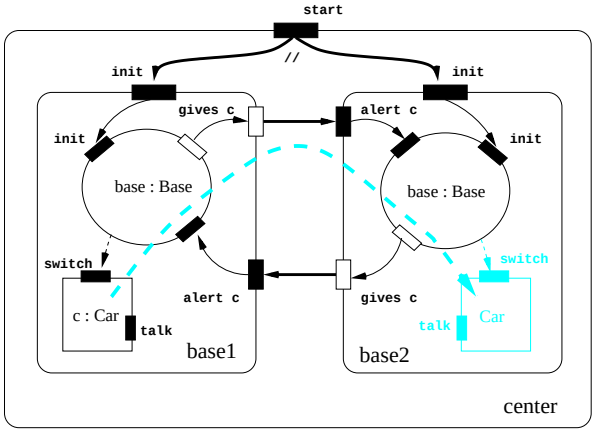


Figure 5. Contextual Model of the Example

3.1. The center context

We model the *center* as a context because it can be naturally thought of as a coordination component. We simply have to describe how to connect the sub-contexts *base1* and *base2* through the gates *gives* and the methods *alert* in order to allow them to exchange information. The following axiom of Figure 6:

```
gives In base1 c With alert In base2 c;
```

synchronizes the event "a car *c* is ready to cross from *base1* to *base2*" with the fact "*base2* is informed that a car *c* arrives". The sequence is implicitly given by the system behavior. The *start* method is an initialization method that must be activated explicitly at system start-up; *start* tells which bases are active or idle at that moment (for the needs of this example, only the initially active base should instantiate a car).

```
Context center;
Interface
Use IdleActive;
Method start;
   Body
      Use Car;
      Use Contexts
         base1,base2;
   Axioms
      start With
         init In base1 active // init In base2 idle;
      gives In base1 c With alert In base2 c;
      gives In base2 c With alert In base1 c;
   Where
      c : car;
End center;
```

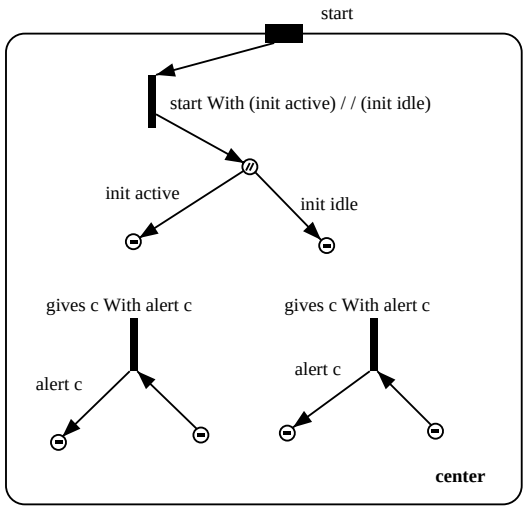


Figure 6. The *center* context coordinating two *base* sub-contexts

3.2. The base contexts

The *base* contexts (i.e. the identical *base1* and *base2* contexts, that have to be defined separately, due to current limitations of the formalism) can be seen as domains that cars cross (under the control of the corresponding radio cell of a cellular mobile telephone system). Each of these contexts encapsulate a "controller" object (a statically created instance of the *Base* class to be described later) which does the actual job of managing the cars inside the domain (see Figure 7 and Figure 8). Therefore, the *Base* class has the same interface as the *base* contexts, and adds other methods for use strictly inside the context (e.g. *talk*). The *base* context axioms simply connect the gates and methods of its interface with the corresponding object of class *Base*.

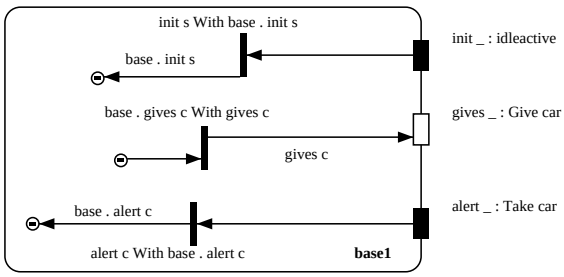


Figure 7. The *base* contexts shielding *Base* locations

3.3. The Base class

This class defines the "controllers" encapsulated inside the *base* contexts. Two kinds of initializations are possible: *active* or *idle*. The method *init* with *active* as argument results in the creation, followed by the initialization, of a new car. The *once* place, of type *blackToken* (its only value is represented as @), ensures that *init* is called only once. If a new car is created, its reference is stored in the *contactcar* place. At this moment, the car knows its contact base (*Self*), and can then chat with it (see Figure 8).

```
Class Base;
Interface
   Use IdleActive, Car;
   Type base;
   Gate gives _ : car;
   Methods
      talk;
      alert _ : car;
      init _ : idleactive;
Body
   Use BlackTokens;
   Place
      contactcar _ : car;
      state _ : idleactive;
      once _ : blackToken;
   Axioms
      init active With c.create .. c.init(Self) ::
         once @ -> state active, contactcar c;
      init idle :: once @ -> state idle;
      alert c With c.switch(Self) ::
         state idle -> state active, contactcar c;
      gives c ::
         state active, contactcar c -> state idle;
   Where c :car;
   Initial once @;
End Base;
```

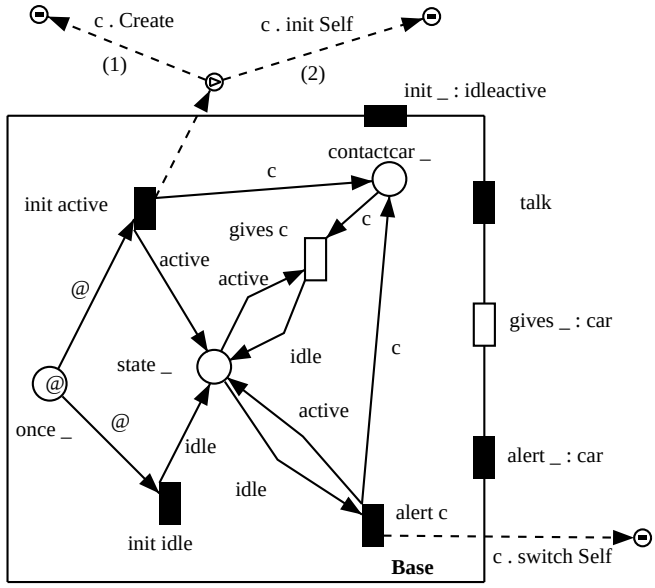


Figure 8. The *Base* class, the instances of which are encapsulated inside *base* contexts.

Migration proceeds as follows, with *base1* initialized to *active*, and *base2* to *idle*. The *gives* gate of *base1* is enabled since its *state* is *active* and a car is referenced in its *contactcar* place. Inside *base2*, the *alert* method is enabled and can be called since the *state* is *idle*. The *gives* gate will then consume the car token from the *contactcar* place, and the external coordination component will fusion the *gives* gate of *base1* with the *alert* method of *base2*. The synchronization of the *alert* method allows to bind the parameter to the migrating car reference, so that the *switch* method of the car may be called to inform it of its updated location. Finally, *base2* receives the car and stores it as a token in its *contactcar* place. The car can then chat through its new contact base.

```
Class Car;
Interface
   Use Base;
   Type car;
   Methods
      talk;
      switch _ : base;
      init _ : base;
Body
   Place contactbase _ : base;
   Axioms
      talk With b.talk
         :: contactbase b -> contactbase b;
      switch b
         :: contactbase b1 -> contactbase b;
      init b :: -> contactbase b;
   Where
      b, b1 : base;
End Car;
```

Figure 9. The *Car* class

3.4. The Car class

The *Car* class illustrates an example of mobile entity in CO-OPN/2; in fact, nothing distinguishes it from any other class. Its *contactbase* attribute contains a reference to the active base which is currently in charge of the car. This attribute is updated by the *switch* method, whereas the old value is ignored. The *talk* method uses this reference in order to identify the particular base the chat is to flow through (see Figure 9, previous page).

3.5. From mobile phones to mobile agents

In the previous model, we have shown how contexts are used for coordination in the mobile telephone example. The notions introduced here allow us to naturally make an analogy between the mobile telephone example and a mobile agent system. The correspondence of elements can be done as follows. The mobile component (the car) naturally becomes an agent. The *base* contexts correspond to the mobile agent execution environments or platforms. The *Base* class represents the effective execution platform (e.g. a virtual machine) which manages the local resources and the agent's migration. Finally the *center* context can be seen as the system or network containing multiple interconnected platforms. In the following sections we first present the additional requirements a modeling formalism needs to meet to encompass the mobile agent paradigm; then we give an overview of our mobile agent model.

4. CO-OPN/2 for Modeling Mobile Agents

When modelling mobile agent systems, we are interested in the expression of the following notions:

- Locality of resources in a broad sense, including code, data, computing power and services.
- Dynamic discovery of such resources.
- Mobility of code alone (i.e. mobile code), or with an associated state (i.e. mobile agents); weak versus strong migration (i.e. whether or not the agent must explicitly manage its state in relation to its movements across execution hosts); active (i.e. initiated by the agent itself) versus passive (i.e. initiated by the environment) migration.
- Security issues, the most relevant being privacy and integrity.
- Communication inside and outside the mobile agent system. Agents are normally expected to be able to do synchronous as well as asynchronous communication with its local and distributed environment (e.g. with its home node). Remote communication should not be neglected, as one cannot expect agent platforms to be available on all hosts; in other words, the formalism should allow modeling complete, i.e. hybrid, distributed systems, combining mobile agent technology and traditional, non-mobile components.

These categories are not necessarily entirely relevant at the level of a specification, when concentrating on functional aspects. Nevertheless, in the current state of research, the main distinction between the mobile agent and the intelligent agent communities is that the former is almost exclusively focused on non-functional aspects, on the contrary of the latter. Many aspects of security, such as the prevention of denial-of-service attacks, are usually too low-level to be considered before the implementation phase. Similarly, the distinction between mobile code and mobile agents (the former being viewed as an implementation-level optimization of the latter) will often be irrelevant when modelling a distributed system. If, however, the goal is to be able to compile the model into an executable program, then it becomes sensible to be able to express somehow such low-level notions.

4.1. Locality and mobility

A simple kind of mobility, which corresponds to *object association* in object-oriented programming, may be easily modelled in CO-OPN/2, as in all Petri net variants including a notion of tokens as object references[1]. CO-OPN/2 may thus manage mobility in the same way as pi-calculus [13]: it is a mobility by reference passing (which could have constituted an initial version when demonstrating the mobile telephone example earlier in this paper). They are equivalent in their way of specifying mobility; however, they differ in the sense that Petri nets are well suited for modeling causality relations between events, while process algebras are better suited for composing processes. This simple way to model mobility can be used to specify distributed object

1. This also means that the language makes it possible to model completely recursive structures, where objects may be nested inside other objects.

systems but is not well adapted for mobile agents since no notion of location boundaries is available: the motivation behind the use of mobile agents is indeed that resources, such as data and services, should often be exploited locally instead of remotely through traditional remote method calls.

The need for locality is one of the reasons why the notion of *context* modules was introduced in CO-OPN/2 with COIL. Contexts provide locality information, and since objects can cross context boundaries, they allow us to model agent migration between contexts. COIL contexts are in fact a kind of protection domain, because no external object can call methods of an object inside a context (even if a reference is available from the outside). All interactions must be performed through special *methods* on the context membrane. Once an object migrates into a context, all objects or contexts inside the receiving context become visible to the incoming object. Until the introduction of COIL, the structuring facilities of CO-OPN/2 had the weakness that they were essentially a tool to define visibility rules, but the operational semantics was not completely modular. Therefore, the other important contribution of COIL is that the operational semantics of CO-OPN/2 is modified to make the language truly modular: locality is now also present in the modelled system's behaviour, which is no longer a result of a flattening of all objects into a single dimension, but a distributed calculus of local behaviours. Thus, contextual coordination, context boundary, locality information, object migration and context hierarchies give us the means to specify mobile code systems.

Strong, weak, active and passive mobility may be modelled in CO-OPN/2. An agent must however be in a stable state (i.e. with no fireable internal transitions) before it can migrate.

Behaviour, on the contrary of object states, is a global information in CO-OPN/2. In other words, it is not possible to move a CO-OPN/2 class *per se* from one context to another; only instances of them may move. As a consequence, mobile code, as opposed to mobile agents, may only be simulated[1].

4.2. Security

Whereas strict object encapsulation is a good step towards the realization of secure mobile agent systems, it should additionally be possible to change the visibility of an agent and its attributes, as well as to supervise its activity and communication according to dynamic policies, as enforced in the Seal Calculus [16] and its implementations (see e.g. the J-SEAL2 framework [4]). We are expecting good results with the modelling of the hierarchical control structures of the Seal Calculus by means of COIL contexts. The notion of context will however need to become entirely dynamic to enable this. Mobile contexts would also be very useful to customize each agent's security policy, whereas now, each agent has to fit into an existing unused agent context when it arrives on a new platform. The actual implementation of our CO-OPN models should be made on a secure platform such as that provided by J-SEAL2 in order to guarantee, in addition to the usual satisfaction of functional properties, some important non-functional properties such as the security of private data and fair access to the platform resources.

4.3. Dynamic discovery of resources

To be faithful to the nature of open and/or large-scale environments such as the Internet, a modeling language should enable the dynamic discovery and management of resources. This is realized to a certain extent by current agent platforms, by adding the ability to express reflexive behaviours, and to manage the above resources as first-class objects. This may currently only be simulated in CO-OPN/2.

5. Towards A Complete Model of Mobile Agent Environment

In the previous, we described the aspects of CO-OPN/2 that are used to model mobility. Now we outline the structure of a mobile agent system, and show the main components of a simplified mobile agent environment described with CO-OPN/2 and COIL. The example describes a system allowing multiple mobile agents to run in an execution platform.

1. We may e.g. define models where class instances are allowed to exist and to migrate in some intermediary, uninitialized state; such objects are then viewed as class surrogates.

5.1. General Architecture

The system is composed of the following classes and contexts (see Figure 10, where the square box represents a class, and the circles are instances of classes):

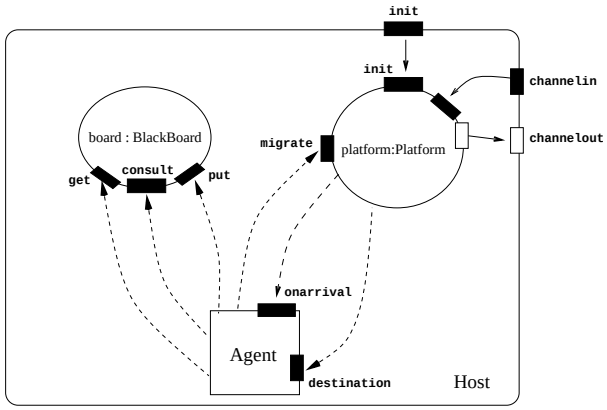


Figure 10. Specification of a *Host* context

- An *Agent* class that specifies the basic structure and behavior of a mobile agent.
- A *Platform* class that represents the actual mobile agent execution environment. This class manages migration of agents.
- A *BlackBoard* class representing a shared data structure where mobile agents can insert, consult or retrieve information.
- A *Host* context representing the environment where mobile agents are executed. This context contains a *platform* and a *blackboard* object, references to which are given to new agents as they enter the host, and this constitutes a model of dynamic linking, as existing in real agent systems.
- A *System* context (not shown in Figure 10) that contains an arbitrary number of connected *Host* contexts; this is a static structure, which means that we can unfortunately not model *Hosts* appearing and disappearing dynamically in the network. *Hosts* have special communication channels for exchanging mobile agents: *channelin* and *channelout*. These will be used for interconnecting the platforms, as specified in the System context.

5.2. Specification of a Basic Agent

The *Agent* class has two methods: *destination* and *onarrival* (Figure 11). The first one is used to ask where the agent wants to migrate, and the second is used to update information on the agent after migration. It also has different places containing: a reference to the current platform (*platform*), a reference to the current blackboard (*blackboard*), a flag indicating the end of the agent activity (*activity*), a flag indicating if the agent is in a migration process (*migration*) and the new destination platform (*nexthop*). Two internal transitions are used to enable the agent activity (*run*) and to trigger migration (*migrate*).

When the *run* transition is fired, the agent will start its main activity. In this simple example, it will get a new platform destination from the current blackboard object. This action ends the agent activity (the *activity* attribute is set to *done*) and the corresponding platform *id* is inserted into the *nexthop* place (the identity is specified as an Abstract Data Type rather than a reference to a *platform* object to simplify the specification). Those conditions will then enable the *migrate* internal transition. This transition is synchronized with the *migrate* method call on the currently enclosing platform object. The agent tells its identity to the platform (*Self*) and the *migration* attribute is updated to *migrating*, so no more activity is possible. This allows the agent to wait in a *stable state* until the end of the migration process performed by the platform. This simple roaming agent class may be redefined in a sub-class in order to specialize its behaviour. The developer will thus be able to concentrate on his agent's added value, since its basic behaviour will be inherited.

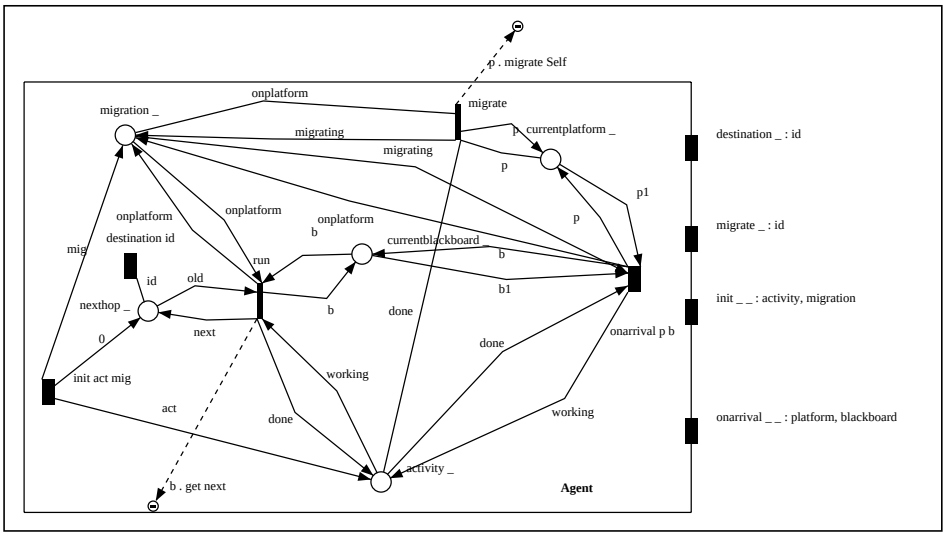


Figure 11. Specification of a primitive mobile agent

5.3. The Platform Class

The Platform Class (Figure 12) has four places containing: the platform's unique id (*id*), a reference to the associated blackboard in the same context (*blackboard*), all active agents running in the platform (*activeagents*), and all agents that are ready to migrate (*readytogo*). The platform has a *migrate* method that inserts the reference of the invoking agent in the *readytogo* place. Through *channelout* it removes from the *activeagents* and the *readytogo* places the reference to the agent requesting to leave. Note that this operation is synchronized with a call to the *destination* method of the requesting agent in order to send the agent to the correct destination platform. Finally the *channelin* method will insert the newly arriving agent into the *activeagents* place and this will be synchronized with the *onarrival* method of the incoming agent in order to update all references and to set the agent as ready to run.

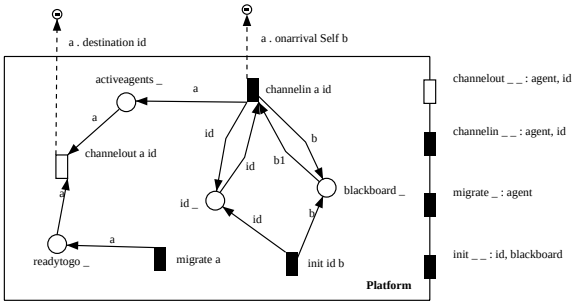


Figure 12. Specification of a mobile agent Platform

6. Perspectives

With this basic architecture, we are able to model very simple agents that migrate over multiple platforms using information stored locally on the different platforms they come across. This shows how mobile agent migration can be specified using CO-OPN/2 and COIL, and is the model to extend in order to be able to model more complex mobile agent environments and applications by way of specialization of agents, platforms and blackboards. We have e.g. experimented with extending and sub-classing the basic *Agent* class in order to obtain different, more useful agent behaviors. By redefining the run transition of *Agent*, so

as to activate a method of a separate *Behaviour* object, we transform the initial primitive agent into a kind of "run-time" support whereas the actual application is modelled inside the *Behaviour* object. Moreover, the primitive Platform class presented here has been cleaned up and completed, by grouping all the information needed by incoming agents (e.g. the references to Self and the blackboard) inside a single *Binding* object.

Several activities are taking place to complement the work described in this paper. We can mention:

- Develop extensions of COIL to provide mobile and dynamic contexts. This would enable the modelling of contexts specific to a given agent, and following it in its migrations, in a manner close to the *ambient calculus* [9].
- A Java code generator to implement mobile agents automatically from their CO-OPN/2 specifications. This work is based on the results described in [7][10] and on the J-SEAL2 mobile agent framework [4] as concrete target platform.
- Transformation of subsets of CO-OPN/2 into CPN-AMI coloured Petri nets [12] in order to formally verify CO-OPN/2 models with the analysis techniques available for classical Petri nets. Currently, only simulation is possible for analyzing CO-OPN/2 models, due to indecidability problems which are similar to those found in all coloured Petri nets which allow an infinite number of colours. Although this is not the purpose of this paper, verification is one of the important uses of formal methods; these aspects will be studied in the future, with emphasis on how to express properties related to the locality and globality of activities.

7. Conclusion

This paper described how it is possible to use the CO-OPN/2 language for the formal specification of distributed applications involving mobile agents. To our knowledge, this is the first proposal for modelling true mobility with structured Petri nets, i.e. consisting not only of dynamic updating of communication channels, but also describing a complete change of the agent's local environment and visibility. Related proposals can be found, e.g. in [1], but focusing more on providing a concise semantics, and less on software development principles, thus resulting in a radically different solution, lacking e.g. the notion of modularity. We refer the reader to the survey performed in [11] for a wider study of formal methods developed for mobile agents. The benefits a software engineer can obtain from our approach are: the safety of a modelling language with well-defined semantics, a strong integration in the development process, as well as a resulting specification with a clear graphical presentation and a structure and semantics close to current implementation paradigms. In this work, we put emphasis on formal design principles; we believe one valuable result is that it will thus be possible to model and validate many new distributed algorithm issues found in mobile agent applications.

Acknowledgements

The authors would like to thank Alex Villazon, Mathieu Buffo, Giovanna Di Marzo and Olivier Biberstein for their valuable contributions. This work was funded by the Swiss National Science Foundation under grants 20-54014.98 and 20-47030.96.

Bibliography

[1] Andrea Asperti and Nadia Busi, "Mobile Petri Nets", Technical Report UBLCS-96-10, Laboratory for Computer Science, University of Bologna, Italy, 1996.

[2] Stéphane Barbey, Didier Buchs and Cécile Péraire, "A Theory of Specification-Based Testing for Object-Oriented Software", in proceedings of EDCC2 (European Dependable Computing Conference), Taormina, Italy, October 1996, Lecture Notes in Computer Science vol. 1150, Springer-Verlag, 1996, pp. 303-320.

[3] Olivier Biberstein, Didier Buchs, and Nicolas Guelfi, ''Object-oriented nets with algebraic specifications: The CO-OPN/2 formalism'', In G. Agha, F. De Cindio and G. Rozenberg, editors, Advances in Petri Nets on Concurrent Object-Oriented Programming and Petri Nets, Lecture Notes in Computer Science. Springer-Verlag, LNCS 2001, pp. 70-127.

[4] W. Binder, “J-SEAL2, A secure high-performance mobile agent system”, in proceedings of the IAT'99 Workshop on Agents in Electronic Commerce, Hong Kong, Dec. 1999.

[5] Didier Buchs and Nicolas Guelfi, “A Formal Specification Framework for Object-Oriented Distributed Systems”, IEEE Transaction on Software Engineering, vol. 26, no. 7, July 2000, pp. 635-652.

[6] Mathieu Buffo and Didier Buchs, “A Coordination Model for Distributed Object Systems”, Proceedings of the Second International Conference on Coordination Models and Languages COORDINATION’97, September 1997, Lecture Notes in Computer Science, vol. 1282, Springer Verlag, 1997.

[7] Didier Buchs and Jarle Hulaas, “Evolutive Prototyping of Heterogeneous Distributed Systems Using Hierarchical Algebraic Petri Nets”, Procs. IEEE International Conference on Systems, Man and Cybernetics, SMC’96, Beijing, China, Oct. 14-17 1996, pp. 3021-3026.

[8] Didier Buchs, Jarle Hulaas and Pascal Racloz, “Exploiting Various Levels of Semantics in CO-OPN for the SANDS Environment Tools”, Tool Presentations, International Conference on Application and Theory of Petri Nets ICATPN’97, Toulouse, France, 1997, 1997, pp. 34-43.

[9] L. Cardelli and A.D. Gordon, “Mobile ambients”, in Proceedings of Foundations of Software Science and Computation Structures (FoSSaCS), European Joint Conferences on Theory and Practice of Software (ETAPS), Springer Verlag, Berlin, Germany, 1998.

[10] Stanislav Chachkov and Didier Buchs, “From Formal Specifications to Ready-to-Use Software Components: The Concurrent Object Oriented Petri Net Approach”, accepted for publication at the International Conference on Application of Concurrency to System Design (ICACSD 2001), Newcastle upon Tyne, U.K., 25-29 June, 2001.

[11] G. Di Marzo, M. Muhugusa, C. F. Tschudin, “A Survey of Theories for Mobile Agents”, World Wide Web Journal, Special Issue on Distributed World Wide Web Processing: Applications and Techniques of Web Agents, pp. 139-153, Baltzer Science Publishers, 1998.

[12] Pascal Estrailler and Claude Girault, “Applying Petri Net Theory to the Modeling, Analysis and Prototyping of Distributed Systems”, in Proc. of the IEEE/SICE Int.Workshop on Emerging Technologies For Factory Automation: State of the Art and Future Directions, Australia, August 1992.

[13] Robin Milner, “The polyadic pi-calculus: a Tutorial”, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK, No. ECS-LFCS-91-180, 1991, Appeared in Proceedings of the International Summer School on Logic and Algebra of Specification, Marktoberdorf, August 1991. Reprinted in Logic and Algebra of Specification, eds. F. L. Bauer, W. Brauer, and H. Schwichtenberg, Springer-Verlag, 1993.

[14] R. Valk, “Concurrency in Communicating Object Petri Nets”, In G. Agha, F. De Cindio and G. Rozenberg, editors, Advances in Petri Nets on Concurrent Object-Oriented Programming and Petri Nets, Lecture Notes in Computer Science. Springer-Verlag, LNCS 2001, pp. 164-195.

[15] C. Lakos, “Object Oriented Modeling with Object Petri Nets”, In G. Agha, F. De Cindio and G. Rozenberg, editors, Advances in Petri Nets on Concurrent Object-Oriented Programming and Petri Nets, Lecture Notes in Computer Science. Springer-Verlag, LNCS 2001, pp. 1-37.

[16] J. Vitek and G. Castagna, “Seal: A framework for secure mobile computations”, In Internet Programming Languages, 1999.

Towards a Unified Approach for Modeling and Verification of Multi Agent Systems

Michael Köhler Heiko Rölke

University of Hamburg, Department of Computer Science
Vogt-Kölln-Straße 30, D-22527 Hamburg
{koehler, roelke}@informatik.uni-hamburg.de

Abstract

For complex software systems an agent-oriented design is assumed. The question of modeling and verification is our research area. In this presentation we outline an unified approach for multi agent systems with respect to modeling and verification.

This general architecture – called MULAN – is formulated in terms of high level Petri nets, namely reference nets. The formalism of reference nets is based on the "nets within nets" paradigm of Valk, which fits well in the context of agent systems.

Here, we first show how multi agent systems should be modeled. In the second step we analyze which are the requirements for a verification systems dealing with multi agent systems. In both cases we arrive at the conclusion that reference nets are well suited to model and verify central aspects of agent systems, like mobility, adaptation, and cooperation.

In this work we focus on modeling and verification of agent conversations. Our approach is based on inter-acting agent protocols. To express our ideas we have chosen the well known producer-consumer scenario both for the model and the formal treatment.

Keywords: modeling, multi agent systems, nets within nets, reference nets, verification.

1 Introduction

Correctness of software is crucial in almost every system. Agent oriented modeling is an established approach to model complex, flexible systems. The proper treatment of encapsulation and composition raises question both in the field of architectural models and compositional verification.

The multi agent architecture MULAN[1] – designed and implemented at our department – is completely modeled with Petri nets (cf. [KMR01]). MULAN is a Petri net based architecture, which is both an implementation of a multi agent platform and also a framework for the modeling of agent applications. The MULAN architecture is based on the "nets within nets" paradigm – formulated by Valk in [Val87] for task flow systems and in [Val96, Val98] for object net systems. MULAN is specified in the reference net formalism (cf. [Kum98]) and implemented with the tool Renew (cf. [KW98]).

Currently, agents are generally programmed using high-level languages such as Java (namely in agent frameworks as Jackal [CFL+98]) or they are defined by simple scripts. A graphical modeling technique that captures all parts of agents and their systems – as UML[2] in the context of object-orientation – is neither proposed nor in general use.[3] An unifying framework based on the visual programming concept of Petri nets – as proposed by MULAN – thus bridges the gab between modeling and programming.

Multi agent systems rely heavily on the mechanism of composition: agent protocols are composed to agent behavior, agent behavior is composed to agents, agents are composed to groups, groups are composed within agent platforms, platforms are composed to agent systems - to name the obvious ones. All these examples leads towards the topic of compositional verification – a central requirement for a proof system for an agent-architecture. Compositionality of proofs requires,

[1] MULAN stands for **Mul**ti **a**gent **n**ets.

[2] UML stands for Unified Modeling Language. See for example [RJB99].

[3] The authors are aware of the upcoming proposals that base on UML i.e. the ones from Odell et al. [BOP00] (AUML). To our opinion these proposals capture only parts of the agent modeling tasks and leave out important areas such as agent mobility.

that a property $spec(S)$ of the composition $S = (S_1||\ldots||S_n)$ is derivable only from the properties $spec(S_i)$ of the components (cf. [Zwi89]):

$$spec(S_1||\ldots||S_n) = f(spec(S_1), \ldots, spec(S_n))$$

We will show, that compositionality in multi agent systems has to be based on the concept of environment: In the system specification of a single module, the environment is considered to be necessarily specified in addition. It is unclear whether the environment should be included in the model (as done e.g. in [DE96]) or whether the module and its environment should be formulated separately but connected in an "assumption-commitment" pattern – a style of reasoning that has been proposed by Chandy and Misra in [MC81].

Our work is structured as follows: section 2 introduces in the MAS-architecture MULAN; in section 3 we describe the general requirements towards a verification system for MAS and how they are met by MULAN; as a result we obtain the necessity to explicitly include assumptions and commitments (A/C) in our framework.[4] Such an explicit notion of A/C is expressed by the formalism of A/C-Petri nets, described in section 4. The formalism is illustrated in section 5 by a formal treatment of the producer-consumer scenario. The work closes with a conclusion.

2 The Mulan architecture

This section gives a short introduction to a multi agent system modeled in terms of "nets within nets". This survey is given to make the general ideas visible that are prerequisite to the understanding of the concepts that follow in later sections of this paper. It is neither an introduction to multi agent systems nor the assets and drawbacks of dividing the system into platforms are discussed here. For a broad introduction see for example [Wei99], the special view taken in our work is a standard proposal of the "Foundation for Intelligent Physical Agents" (FIPA) [FIP98a]. The latest publications of the FIPA can be found in [FIP].

2.1 Reference nets

MULAN is based on the special Petri net formalism of reference nets [Kum98].[5] As for other net formalisms there exist tools for the simulation of reference nets [KW98]. Reference nets show some expansions related to "ordinary" colored Petri nets: nets as token objects, different arc types, net instances, and communication via synchronous channels. Beside this they are very similar to coloured Petri nets as defined by Jensen [Jen92]. The differences are now shortly introduced.

Nets as tokens Reference nets implement the "nets within nets" paradigm of Valk [Val96, Val98]. This paper follows his nomenclature and denominates the surrounding net *"system net"* and the token net *"object net"*. Certainly hierarchies of net within net relationships are permitted, so the denominators depend on the beholder's viewpoint.

Arc types In addition to the usual arc types reference nets offer *reservation arcs*, that carry an arrow tip at both endings and reserve a token solely for one occurrence of a transition, test arcs and inhibitor arcs. Test arcs do not draw-off a token from a place why a token can be tested multiple times simultaneously, even by more than one transition (test on existence). Inhibitor arcs prevent occurrences of transitions as long as the connected places are marked.

Net instances Net instances are similar to the objects of an object oriented programming language. They are instantiated copies of a template net like objects are instances of a class. Different instances of the same net can take different states at the same time and are independent from each other in all respects.

[4] The notion of "assumptions and commitments" does not rely directly on the original definition of Chandy and Misra, but rather denotes a style of reasoning.

[5] It is assumed throughout this text that the reader is familiar with Petri nets in general as well as coloured Petri nets. Reisig [Rei85] gives a general introduction, Jensen [Jen92] describes coloured Petri nets.

Synchronous channels Synchronous channels [CH94] permit a fusion of transitions (two at a time) for the duration of one occurrence. In reference nets (see [Kum98]) a channel is identified by its name and its arguments. Channels are directed, i.e. exactly one of the two fused transitions indicates the net instance in which the counterpart of the channel is located. The other transition can correspondingly be addressed from any net instance. The flow of information via a synchronous channel can take place bidirectional and is also possible within one net instance.

2.2 Architecture

Take a look at figure 1: The gray rounded boxes enclose nets (net instances) of their own right. The ZOOM lines enlarge object nets that are tokens in the respective system net.[6] The upper left net of the figure is an arbitrary agent system with places containing agent platforms and transitions modeling communication channels between the platforms.[7]

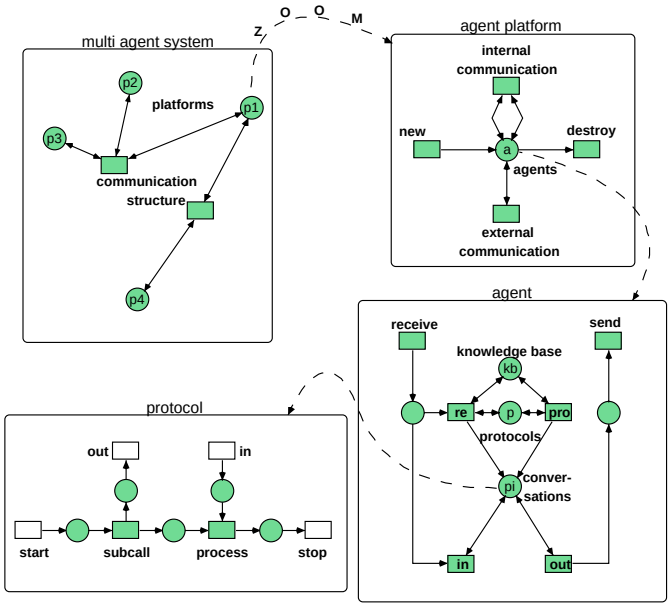


Figure 1: MAS as nets within nets

The first zoom leads to a closer view of a simplified agent platform. The central place agents contains all agents that are currently hosted on the platform. New agents can be generated (or moved from other platforms) by transition new, agents can be destroyed or migrate to another platform (transition destroy).

Internal message passing differs from the external case – so it is conceptually separated: The internal communication transition binds two agents (the sender and the receiver of a message) and allows them to hand over a message via call of synchronous channels. External communication involves only one agent of the platform. For external communication as well as for agent migration the communication transitions of the top level agent system net are needed. The interaction of the multi agent system and the agent platform is made possible by inscribing the transitions with synchronous channels connecting for example the transition external communication of an agent platform with that of another one via the communication structure transition of the multi agent system. These inscriptions are not visible in the figure.

[6] Beware not to confuse this net-to-token relationship with place refinement.

[7] This is just an illustrating example, the number of places and the form of interconnection has no further meaning.

The remaining nets that show the structure of an agent and an example of its (dynamic) behavior in form of protocols (protocol nets) are explained in more detail in the following sections.

Each zoom describes one central concept of multi agent systems. The relationship of the agent system to the platforms raises needs for the concept of mobility, whereas the relationship between the platform and the agent has to be treated by the concept of cooperation. The relationship of agents and their protocols is captured by the concept of adaptation of intelligent agents.

2.3 Agents

An agent is a message processing entity, that is, it must be able to receive messages, possibly process them and generate messages of its own. In this context it is to be noted that a completely synchronous messages exchange mechanism as it is used in most object oriented programming systems, frequently violates the idea of autonomy among agents.[8]

The introduced basic agent model implies an encapsulation of the agents: regardless of their internal structure, access is only possible via a clearly defined communication interface. In figure 2, this interface is represented by the transitions receive message and send message. In the figure, the realization of the interface (through connection of both transitions to a messages transmission network via synchronous channels) is not represented. Several (then virtual) communication channels can be mapped to both transitions.

The presented agent model corresponds to the fundamental assumptions about agents: Because agents should show autonomy, they must have an independent control over their actions. Autonomy implies the ability to monitor (and, if necessary, filter) incoming messages before an appropriate service (procedure, method...) is called. The agent must be able to handle messages of the same type (e.g. being asked for the same service) differently just because of knowing about the message's sender. This is one of the major differences between objects and agents: A *public* object method can be executed by any other object, *protected* methods offer only a static access control that is very often inconvenient to the programmer and user.

The processing of messages is realized by a selection mechanism for specialized subnets, that implement the functionality of the agent, therefore (beside the selection process) its behavior. These subnets are named *protocol Petri nets* (or short *protocols*) in the following.

Each agent can control an arbitrary number of such protocols, possesses however only one net (in reference net nomenclature: one net page), that represents its interface to the agent system and therewith its identity. This main net (page) is the visible interface of an agent in the multi agent system. As mentioned before all messages that an agent sends or receives have to pass this net.

The central point of activity of a protocol-driven agent is the selection of protocols and therewith the commencement of conversations [CCF+99, KMR01]. Conversation consists of a set of protocols spread over several cooperating agents. The protocol selection can basically be performed pro-actively (the agent itself starts a conversation) or reactively (protocol selection based on a conversation activated by another agent).[9] This distinction corresponds to the bilateral access to the place holding the protocols (protocols in conversation). The only difference in enabling and occurrence of the transitions reactive and pro-active is the arc from the place incoming messages to the transition reactive. So the latter transition has an additional input place: the incoming messages buffer. It may only be enabled by incoming messages. Both the reaction to arriving messages and the kick-off of a (new) conversation is influenced by the knowledge of an agent. In the case of the pro-active protocol selection, the place knowledge base is the only proper enabling condition, the protocols are a side condition. In simple cases the knowledge base can be implemented for example as a subnet, advanced implementations as the connection to an inference engine are also possible (and have been put into practise). Unfortunately this topic cannot be deepened here any further.

[8] To our understanding agents are not exclusively (artificial) intelligent agents, but rather a general software structuring paradigm on top of the ideas of object orientation [Jen00].

[9] The fundamental difference between pro-active and reactive actions is of great importance when dealing with agents. An introduction to this topic is e.g. given by Wooldridge in [Woo99].

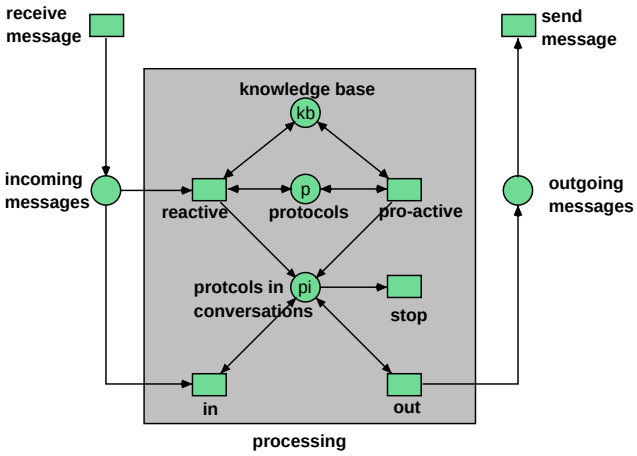


Figure 2: A protocol-driven agent

A selected and activated protocol[10] is also called a conversation-part because it usually includes the exchange of messages with other agents. A conversation can however also run agent internal, therefore without message traffic. A freshly invoked conversation gets assigned an unambiguous identification that is not visible in the figure. All messages belonging to a conversation carry this identification as a parameter to assign them properly. If an agent receives a messages carrying such a reference to an existing conversation, transition in is enabled instead of transition reactive. The net inscriptions that guarantee this enabling are not represented in figure 2 for reasons of simplicity. The transition in passes incoming messages to the corresponding conversation protocol in execution. Examples for this process follow in section 2.4.

If the sending of messages to other agents is required during the run of a conversation, these messages are passed from the protocol net over the transition out to the agent's main page and are handed over to the message transport mechanism by the transition send message.[11] The communication between protocol net (conversation) and the agent's main net takes place via synchronous channels.

An interesting feature of any agent derived from the (template) agent in figure 2 is that they cannot be blocked, neither by incoming messages nor by their protocols[12] and therefore cannot loose their autonomy.

Examples for concrete conversation protocols are to be found in the following chapter, where a producer-consumer process is modeled exemplarily. This scenario acts also as the case study for our verification approach for MULAN.

2.4 Protocols

An important field of application of Petri nets is the specification of processes such as that in figure 3, that shows a simple producer-consumer process. In order to give no room to conceptual confusion, such nets that spread over several agents and/or distributed functional units will be called "survey nets".

The place buffer in the middle of the figure represents an asynchronous coupling between the process of producing and that of consuming. This coupling is however to that extent dependent that it for example blocks the consumer if it is empty or, given the case that it is inscribed with a capacity, blocks the producer when this maximal filling is reached. In the following, producer and

[10] Following the object oriented nomenclature one speaks of an instantiated net or protocol (that is represented in form of a net).

[11] The message transport mechanism is part of the agent system (or platform) and is therefore only sketched in this section.

[12] Unless it is strictly necessary for a protocol to block the entire agent and this is explicitly modeled.

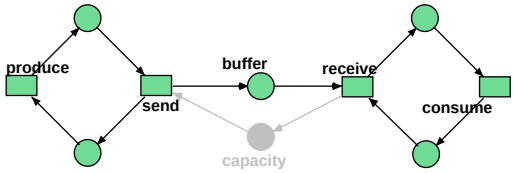


Figure 3: Producer-consumer (survey net)

consumer are introduced as autonomous agents and are modeled according to figure 2 by means of a reference net. The buffer is not modeled as an independent agent, nevertheless this would both syntactically (this will be explained in the following) and semantically (in consideration of the level of autonomy the buffer owns) be no problem.

An interesting point is the re-usability of the protocols: Consider a refined model in which the buffer should play an active role and should therefore be modeled as an agent of its own. The protocols of the producer and the consumer remain structurally unchanged, only the addressees of their messages have to be adapted. But these should be modeled dynamically in any case.

The following example assumes that the buffer is restricted by a capacity of one item. This restriction is for simplification purposes only and may be lifted easily. The restriction is indicated in figure 3 by the gray place capacity under the buffer place.

Producer The protocol of the producer agent is represented in figure 4. The upper transitions with the channels :start, :out, :in, and :stop are typical for all types of protocol nets. The :start channel serves as a means to pass possibly necessary parameters to the protocol. It is called on the agent main page (see figure 2) either by transition reactive or pro-active. The channels :in and :out are responsible for the communication of an operating protocol with the environment. They connect to the transitions of the same denominators on the agent's main page. When a protocol has finished its task, the transition inscribed with channel :stop is enabled. By calling of this channel the agent may delete the protocol or, more correctly, the protocol instance.

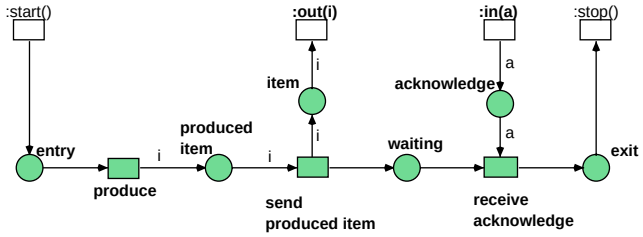


Figure 4: Producer protocol

After the start of the protocol the transition produce produces a performative[13] (here i) containing an item, that is directed to the consumer. Note that in the example the performative is the only thing that is produced. The performative will be sent over the :out channel; subsequently the protocol is blocked waiting for an answer message. The blocking behavior is necessary to simulate a synchronous communication between producer and buffer. Without waiting for an answer the producer would be able to "inundate" the buffer with messages, what requires an infinite buffer capacity. An arriving confirmation enables the transition receive acknowledge. After occurrence of that transition the protocol is not blocked any further and terminates (by enabling the stop transition). The producer agent is now able to select and instantiate the produce protocol again.

[13] Some of the ideas that led to the agent model introduced here are partially originated in the area of the KQML-([FL97]) or FIPA-agents ([FIP98b]). Roughly speaking a performative is a message. KQML stands for "Knowledge Query and Manipulation Language", FIPA is the abbreviation of "Foundation for Intelligent Physical Agents".

Consumer The protocol net that models the consume behavior of the consumer agent (see figure 5) is selected (*reactively*) by the agent's main page to process an incoming performative from the producer agent. It is instantiated and the :start channel is used to pass the performative to the protocol. Beside others the performative is needed to send an acknowledge performative to the originator of the conversation (the producer). Note that the consumer agent does not know the producer or if there is one or several of them. The protocol works in either case.

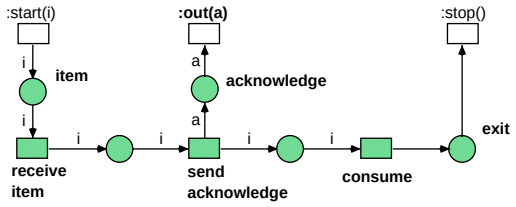


Figure 5: Consumer protocol

After receiving the item and sending the acknowledge the transition consume may occur. After that the protocol terminates and can be deleted.

Figures 4 and 5 show the protocols that model a conversation between producer and consumer. They are executed within agents of the type of figure 2. The figures form a simple example that illustrates how to model a producer-consumer process by means of agent oriented Petri nets. The proposed methodology to implement protocol nets in a top-down manner starting with so-called survey nets is not the only possibility to develop protocols. One can easily think of a bottom-up style or mixed cases especially for hierarchical protocol relationships. Unfortunately this topic can not be deepened here.

3 Agent oriented verification

The analysis of agent systems raises the need for a special style of verification systems. This need is due to the dynamics and openness of agent systems. As stated before, modeling approaches lack a uniform basis. The same is true also for verification approaches.

Most approaches handle multi agent systems (MAS) the same way as a single intelligent agents. This style is based on the traditional artificial intelligence (AI) view, disregarding the needs of MAS. It is mostly based on (modal) logical specification and model checking techniques (cf. [HR00] as an example).

In general, approaches addressing the specialties of MAS focus on one single aspect of description. For mobility several algebraic approaches describe the change of environment (cf. [MPW92], [VC99] or [CGG99]). Adaptation (cf. [GPdFC98]) and cooperation (cf. [SHM99] and [CCF+99]) are addressed also.

In our point of view such approaches are insufficient for the analysis of an agent system, since only parts of the system are specified. It is unclear whether the combination of the isolated formalizations leads to a correct description of what the system does. Since the integration of the models has do be done manually by the developer this leads to an error pruning style of construction.

So, we conclude that not only the construction of an agent system should be done on a uniform basis but also the formal part of reasoning about it should be based on one single calculus. In the following, we describe which characteristics such a formal basis should own.

In our MULAN-approach, we understand the central parts of a dynamic, open agent system – mobility, adaptation, and cooperation – as parts of one central concept: compositionality.

Compositionality as the leading paradigm for agent systems Agent systems have several specialties – all based on the paradigm of compositionality:

- Mobility is naturally expressible as the composition of an agent with its environment. Mobility focus this point towards the dynamic change of environments during run time of an agent system.

  Compositionality is also central for the description of agent groups and their mobility: agents are composed to groups, which can be considered as agents again.

- Cooperation of agents is also a kind of composition, at least at the technical level of interaction. Since the behavior of one single agent is expressed by a set of protocols, conversations are formed by the composition of several agent protocols.

- Adaptation is the dynamic change of agent behavior. Since behavior is a composition of several primitive actions, adaptation must be described by the dynamic configuration of such an composite behavior. So, behavior is a kind of composition phenomena.

These kinds of compositionality have been shown in figure 1 before: each zoom stands for one composition mechanism.

3.1 Classification of verification approaches

The great number of publications in the very general field of verification raises the need for a classification scheme. We propose a classification scheme mainly oriented on two categories: first the type of program being verified and second the verification time and style.

The first category discriminates by the kind of model which is verified

1. *Sequential block.* The program is considered to be a monolithic block, usually build by iteration, conditionals, and sequence. Central work is the approach by Hoare [Hoa69] for algol-like programming languages.

2. *Parallel block.* The second class also deals with monolithic programs, but allows the construct for parallel execution. The most known approach for parallel programs is due to Owicki and Gries [OG76].

3. *Top-down development.* The third class incorporates the aspect of information hiding. Programs are considered to be the composition of encapsulated modules. This style of development and verification is known as the "top-down" approach. The most common proof system has been developed by Apt, Francez, and de Roever [AFdR80].

4. *Bottom-up development.* The fourth class considers the verification of modules on their own. The context where modules are embedded in is not known a priori. This style of reasoning is known as the "bottom-up" approach. A central step in the development of bottom-up verification systems is the assumption/commitment formulation by Chandy and Misra [MC81].

While the first category deals with the model, the second category discriminates by the moment verification takes place and the technique being used:

1. Verification is done *after* programming. First the system is specified, then coded and afterwards the implementation is checked in terms of the specification. Central approach is the model checking technique by Clarke and Emerson [EC82].

2. Verification is done *while* programming. This could be done if the programmer has a central idea how to proof the implementation while he programs. This kind of style is known as "structured programming". All axiomatic approaches – like [Hoa69] – are examples.

3. Proof *by* construction. Programming and verification are essentially the same. These approaches are based on constructive logic (like COQ [BBC+99]) or on property preserving transformation (like in [Ber87]) or on structural restrictions resulting in subclasses (like in [BT87]).

This classification is oriented on the historical development of programming styles and major verification styles. This scheme is illustrated in figure 6.

	sequential	parallel	top-down	bottom-up
post	model checking on a pre-described system	model checking on a pre-described system	"modular" model checking	./.
while	axiomatic: Hoare style	interference test: Owicki/Gries	global invariants, Unity, temporal logic	assumption / commitment style
by	constructive logic, transformation, subclasses	liveness preserving composition	./.	./.

Figure 6: Classification scheme for verification approaches

3.2 From compositional verification to agent oriented verification

Which approaches are suited for the development and verification of agent systems? In the following the central requirements of MAS are contrasted with the existing verification styles, as classified before. We will show that several approaches relevant in the "normal" case are unsuited for the special case of agent systems.

Several approaches try to lift verification styles designed for object oriented programs up to the agent context – neglecting the special needs of multi agent systems.

The first speciality is due to the nature of agent: agents are encapsulated, autonomous entities which are loosely coupled. They are developed without any knowledge of the whole agent system. Developers and agents cannot know the whole system or the state of the whole system. There is no such thing as global knowledge for agents.

Due to encapsulation and isolated development only bottom-up verification can apply to the agent context. All approaches considering agent systems as one unit must reject the openness of agent systems and rely on some kind of "closed agent world assumptions". Only the assumption based style of reasoning can apply to open systems.

Since agents are considered as active entities (in contrast to passive objects), we additionally take into account who is reasoning about the agent system. If it is the developer, we are mainly confronted with "static" problems, if it is the agent, we are confronted with more "dynamic" aspects. In the second case reasoning must be done automatically. This seems to be harder than the first case, where it can be done semi-automatically.

Reasoning at run-time is the consequence when dealing with open systems, where no one can know in advance, what might enter the system. Security in mobile agent systems is a central aspect (cf. [Vig98]) which can only be achieved by structural restrictions on agents or on agent behavior. Structural restrictions reduce the proof burden of an agent system.

If one compares these requirements with the category scheme in figure 6, one can recognize, that we can restrict our investigation to the "bottom-up" column and the rows "verification while and by development" (see figure 7).

	sequential	parallel	top-down	bottom-up
post	–	–	–	–
while	–	–	–	relevant for the developer
by	–	–	–	relevant for the agent

Figure 7: Verification in the agent context

Besides these major issues, which reject several approaches in advance, some additional aspects must be handled in verifications systems in the agent context. These aspects might be of interest in the context of object-orientation, too, but in the context of agents they cannot be ignored.

1. *Encapsulation and modularity.* A proof system must be able to express the concept of information hiding, so it cannot be formulated in a "flat" and global way.

2. *Concurrency.* Agent systems are highly independent and run concurrently. So specification, implementation, and verification should not rely on totally ordered action sequences. Partial order semantics (true concurrency) should be used instead.

3. *Dynamic environment.* Agent systems are conceptually based on distribution and mobility. Therefore, a proof system must be able to describe environments and their dynamic change in an explicit way.

4. *A/C-concepts.* An approach for MAS should allow to specify assumptions and commitments. We additionally postulate, that these assumptions and commitments should not only be visible for the verifier but also for the developer. To avoid a gap between verification and modeling, assumption and commitments should be an integral part of the model.

5. *Structural properties.* The systems should allow to easily describe structural restriction in order to guarantee e.g. security properties. These restrictions should be easily adaptable in the modeling approach.

Due to these requirements we have chosen the paradigm of "nets within nets" [Val87, Val96, Val98] for the developmental and formal basis of the MULAN-architecture. This approach meets the above mentioned requirements directly or is adjusted currently by the authors. It has also the benefit that the "nets within nets" paradigm has it native "machine" implemented in the Renew tool [KW98].[14]

This approach meets the requirements for an unified approach. Like every Petri net formalism it incorporates the concept of concurrency. The requirement of encapsulation and modularity is fulfilled by the concept that nets could be regarded as tokens again – a view that meets well with modularity. The remaining requirements – dynamic environment, A/C modeling and structural guaranteed properties – are captured by the formalism of assumption/commitment Petri nets, short: A/C Petri nets. The subclass of A/C Petri nets, that we are presenting here, deals with protocols and is therefore called A/C-protocols.

The concepts – mobility, adaptation, and cooperation – are supported by concepts on their own as shown in figure 8, where we have shown the relations between the distributed artificial intelligence (DAI) paradigms, the MULAN concepts, and the corresponding entities in the proof system. These concepts cannot be explained in detail, so we sketch briefly the ideas: "Distributed markings" (cf. [Köh00b]) describe the mobility aspect of "nets within nets" in an algebraic way. They are especially suited for the formalization of agent groups in a distributed system. Adaptation is based on the reconfiguration of composed A/C-Petri nets. Cooperation takes place in form of conversations, consisting of ordered A/C agent protocols.

DAI	MULAN	verification
mobility	location nets, group agents	distributed markings
adaptation	self-modifying protocols	A/C-Petri nets
cooperation	conversation	conversation orders, A/C-protocols

Figure 8: Relationship of DAI paradigms and verification of MULAN models

These concepts are based on the algebraic viewpoint on nets ("Petri nets are monoids" [MM90]) which we addressed in former work for processes in object net systems (cf. [Köh00a]). These dependencies – and some more, that cannot be deepened here – are illustrated in figure 9.

In the further presentation we will focus our attention mainly on the aspects of the A/C modeling and verification of agents protocols in terms of A/C-Petri nets (cf. the right part of figure 9).

[14] The work of [BDM+99] favors an approach based on linear logic instead. This work is a good argument for our approach, since Petri nets and linear logic are strongly related (cf. [Far00]). Our approach has the additional advantage that our models also have a graphical representation.

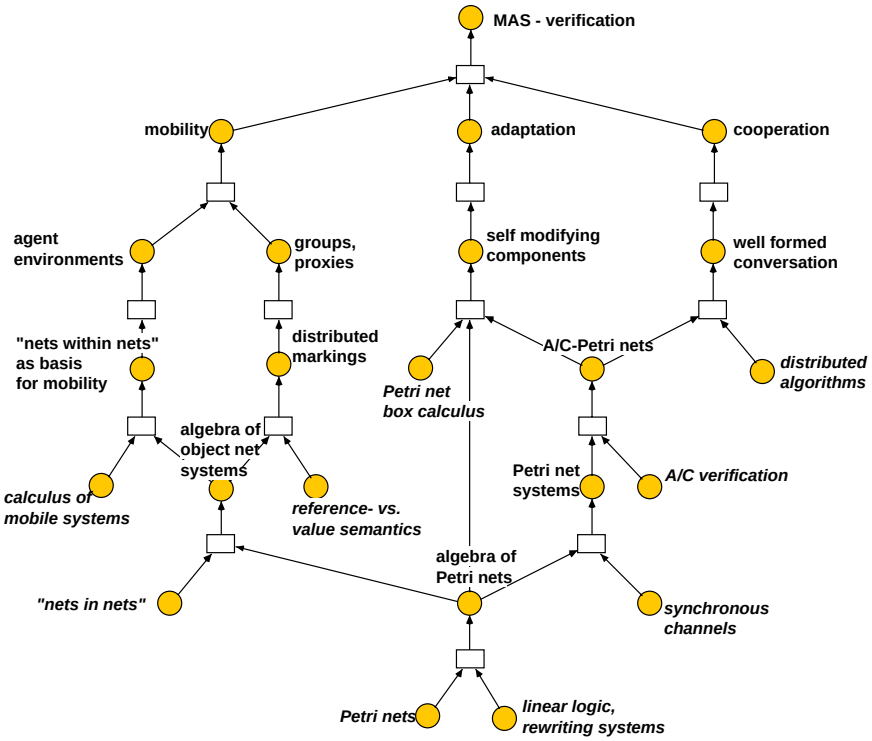


Figure 9: Foundations of MAS verification

4 A/C Protocols

Our formal description of agent protocols is based on the explicit notion of assumptions and commitments (A/C) made towards the environment. Our general model to express A/C is the formalism of A/C-Petri nets. Starting from these A/C Petri nets we define the so called classes of basic protocols Petri nets (BPPN) and of environment based protocols Petri nets (EPPN).

4.1 A/C-Petri nets

An A/C-Petri net is a net $N = (S, T, T^A, F)$, where the subset $T^A \subseteq T$ denotes transitions modeling environmental actions (A stands for assumptions). The set $T^0 = T \setminus T^A$ are the "core" transitions of the net. Have a look at figure 10 for an example: the unfilled transition $t_2 \in T^A$ denotes an environmental transition, while $t_1, t_3 \in T^0$ are core transitions.

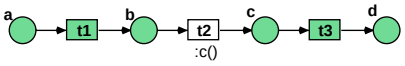


Figure 10: An A/C-Petri net

Environmental transitions $t_A \in T^A$ are not enabled on their own, since it is assumed that t_A must fire synchronously with some transition t_U from the environment U. Because of this, an environmental transition $t_A \in T^A$ is never enabled without the environment, if we consider the A/C-Petri net on its own.

Environmental transitions are modeled by transitions in combination with the concept of synchronous channels.[15] In the example of figure 10 t_2 is inscribed by the up-link :c(). This approach

[15] Note, that synchronous channels are directly supported by the Renew tool. So, A/C-Petri nets could easily be implemented.

has the advantage to allow the modeler to denote explicitly the assumptions towards the environment in the same model. This has a major advantage over the standard approach to fuse protocols (without the explicit notion of assumptions) with a given environment and stating the correctness of the whole model afterwards. In our approach the protocol P^0 is modeled together with the assumptions P^A (resulting in the A/C protocol $P^{A/C}$). This protocol $P^{A/C}$ is fused independently later on with an environment U. Since the assumptions of a protocol are defined in advance, we only have to check, whether a given environment is able to fulfill them.

A/C based verification Interference-free progress is a property of special importance. Interference-free progress is described by the operator $\triangleright$. The formulae $N :: m_1 \triangleright m_2$ describes that from the marking m_1 the marking m_2 is guaranteed to be reached (by steps of N) and during this, additional markings m do not interfere with the sub-process of m_1.[16] So, each property $m_1 \triangleright m_2$ is monotone wrt. multi set addition, so $m_1 + m \triangleright m_2 + m$ is valid, too.

This property meets well with the structure of protocol nets. Since parts of protocols are disjoint in terms of the underlying net graph, processes will be disjoint, too. A special feature of $\triangleright$, that is used in this contribution, is the substitution rule. Let P, Q, R and S be expressions denoting multi-set markings of places, then we have:

$$\frac{\begin{array}{llll} N :: & P & \triangleright & Q + R \\ N :: & R & \triangleright & S \end{array}}{N :: \quad P \quad \triangleright \quad Q + S} \quad \text{(substitution)}$$

Have a look at the net N in figure 10 again: There $N :: a \triangleright b$ and $N :: c \triangleright d$ are valid, since t_1 and t_3 establishes the formulae. But $N :: b \triangleright c$ is not valid, since the transition t_2 is an environmental one, only activated if the channel $:\mathsf{c}$ is activated by the environment.

We now want to specify the properties of an A/C-Petri net $N^{A/C}$ not in isolation, but in the context of possible environments. So we have a look at the family of "embedded nets" $\lceil N^{A/C} \rceil$. These nets are obtained by treating some (or all) environment transitions $t \in T^A$ as simple ones: If $N = (S, T, T^A, F)$ is an A/C-Petri net, then $\forall \emptyset \subseteq E \subseteq T^A : (S, T, T^A \setminus E, F) \in \lceil N^{A/C} \rceil$.

This construction describes the embedding in an environment, which is able to synchronize with some (or all) actions, that are assumed by $N^{A/C}$. A property ψ of the A/C-Petri nets $N^{A/C}$, which is true for every embedding in a net of $\lceil N^{A/C} \rceil$ (syntactically denoted as $\lceil N^{A/C} \rceil :: \psi$) is therefore true for the composition $N^{A/C} || N'$ of $N^{A/C}$ with any environment N'. This rule is called the "embedding rule":

$$\frac{\lceil N^{A/C} \rceil \quad :: \psi}{N^{A/C} || N' \quad :: \psi} \quad \text{(embedding)}$$

In our example the formulae remain valid under embedding: $\lceil N \rceil :: a \triangleright b$ and $\lceil N \rceil :: c \triangleright d$. Note, that $\lceil N \rceil :: b \triangleright c$ is not valid, since it is invalid in one embedding, namely the inactive one.

Synchronous channels Without going into detail[17] we describe the notion of a synchronous channel in terms of a rewriting system. Transition could be labeled with at most one up-link (denoted as $:\mathsf{up}$ if it exists) and a – possible empty – set of down-links, denoted as $\{:\mathsf{down}_1, \ldots, :\mathsf{down}_n\}$.

A transition t is denoted by the operator $(\cdot ? \cdot ! \cdot)$ as $(t ? :\mathsf{up} \, ! :\mathsf{down}_1, \ldots, :\mathsf{down}_n)$. The special case of a transition without an up-link is denoted as $(t ? - ! :\mathsf{down}_1, \ldots, :\mathsf{down}_n)$, the case of zero down-links analogously as $(t ? :\mathsf{up} \, ! -)$, no channels as $(t ? - ! -)$.

The semantics of the net system is described by a concurrent transition system. Two transitions with matching up- and down-link ch could be fused dynamically: $t_1 /\!\!/_{ch} t_2$. This fusion discharges the channel $:\mathsf{ch}$.

$$\frac{\begin{array}{rll} (t_1 ? :\mathsf{ch}_{\mathsf{up}} \, ! (:\mathsf{ch} \, + CH_1)) : & \sum s_i & \to \sum s_j \\ (t_2 ? :\mathsf{ch} \, ! CH_2) : & \sum s_k & \to \sum s_l \\ CH_1 \cap CH_2 = \emptyset & & \end{array}}{((t_1 /\!\!/_{ch} t_2) ? :\mathsf{ch}_{\mathsf{up}} \, ! (CH_1 + CH_2)) : \quad \sum s_i + \sum s_k \quad \to \sum s_j + \sum s_l}$$

[16] Interference-free progress wrt. additional tokens makes $\triangleright$ more restrictive than conventional leads-to properties, since it relates progress with causality.

[17] The complete description of the underlying algebra can be found in [Köh01].

The requirement $CH_1 \cap CH_2 = \emptyset$ prevents cyclic dependencies.

Not all transitions of the transition system are relevant for the semantics. A transition t is included in the semantics of the systems – denoted by the Greek letter τ – if it has no remaining links:

$$\frac{(t?-!-): \quad \sum_I s_i \quad \rightarrow \quad \sum_J s_j}{\tau: \quad \sum_I s_i \quad \rightarrow \quad \sum_J s_j}$$

A special case we are faced with is the fusion of two transitions $(t_1?{:}\mathsf{up}_1\,!-)$, $(t_2?{:}\mathsf{up}_2\,!-)$ by a third one $(t_3?-!{:}\mathsf{up}_1\,,{:}\mathsf{up}_2\,)$ having exactly these two down-links. This special case is denoted by $t_1 \sim t_2$, if t_3 has empty pre- and post sets. $t_1 \sim t_2$ models a symmetric synchronization, which we use to express CSP-like channels.

4.2 Basic Protocol Petri nets

In figure 11 the frame of a basic protocol Petri net (BPPN) is shown. A basic protocol has a starting point (entry), and an exit point (exit), as well as places for each incoming message (in Msg_i) and outgoing ones (out Msg_j). The inner structure of a BPPN is abstracted away, only the interface is shown here.

Let $P.I$ denote the index set of incoming messages, $P.J$ for outgoing ones. It should be able to treat the component as one single transition, that is independent from the outside: Each marking $m = \mathsf{P.entry} + \sum_{P.I} \mathsf{P.in_i}$, denoting the initial state for the BPPN leads to a marking where the final marking $\mathsf{P.exit} + \sum_{P.J} \mathsf{P.out_j}$ is reached:[18]

$$P :: \mathsf{P.entry} + \sum_{P.I} \mathsf{P.in_i} \triangleright \mathsf{P.exit} + \sum_{P.J} \mathsf{P.out_j}$$

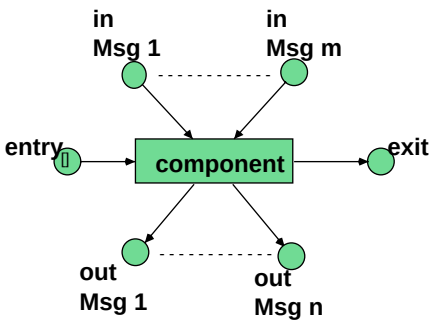


Figure 11: A basic Petri net protocol

BPPN are well suited to specify the behavior of a "server protocol", since it reacts towards incoming messages with outgoing ones. They are not suited to specify a "client protocol" which works correctly only if an outgoing request message is replied by a server – a situation we are faced with in the producer-consumer scenario: so, we need a way to specify that a new item should only be produced by the producer, if the last one is confirmed by the consumer.

If we have a look at figure 4 and 5, it is not clear at the first moment where the fundamental difference is between them, since at the interface-level they look almost the same. So we have to lift some of the information up to the interface-level. This is done in the formalism of environment based protocols (EPPN).

4.3 Protocol nets with environment assumptions

Assumptions, that are fundamental for the correctness of a protocols, should be included in the model itself and should be visible to the developer. These assumptions towards the environment

[18] Note that in the general case only one input message and also only one output message is needed, since having several messages means synchronization. A single message can of course contain several parameters.

are described in terms of the "assumption/commitment" paradigm (inspired by [MC81]). We distinguish four different kinds of messages to express assumptions and commitments: in-, out-, request- and reply-messages (cf. figure 12):

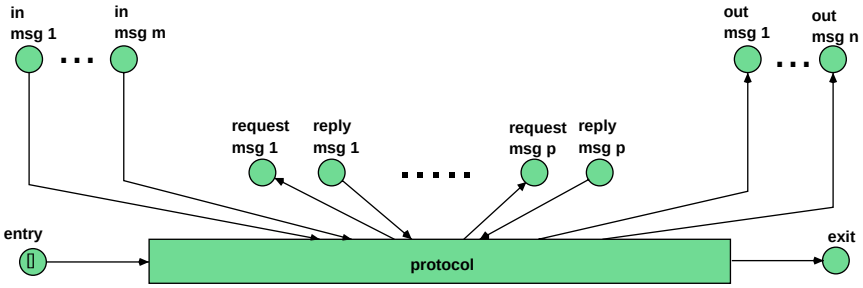


Figure 12: Protocol P^0 with four types of messages

1. Incoming messages, which can be considered as parameters (in $\mathsf{msg_i}$, $0 \leq i \leq m$).
2. Outgoing messages, which are the reaction towards the incoming messages (out $\mathsf{msg_j}$, $0 \leq j \leq n$).
3. Outgoing messages, which are assumed to be replied (request $\mathsf{msg_k}$, $1 \leq k \leq p$).
4. Incoming messages, which are assumed to be a reply towards a former request (reply $\mathsf{msg_k}$, $1 \leq k \leq p$).

Such a protocol net is called environment based protocol Petri net (EPPN). An EPPN integrates the assumptions made by the protocol. The assumption is, that the environment will connect each request-place with a reply place. Thus, the correctness for an EPPN can be checked by connecting each request to a reply place via an environmental transition. This integration enables us to describe correctness of a protocol under the consideration of environment assumptions.

This construction yields in another protocol net P^U. The protocol P^U is obtained if a set of transitions $\{u_1, \ldots, u_p\}$ is added, so that each request-place $\mathsf{request_k}$ is connected by u_k with the corresponding reply-place $\mathsf{reply_k}$ (cf. figure 13). These transitions u_k model the environment U, modeled as the net $U = (\{\mathsf{request_k}, \mathsf{reply_k} : 1 \leq k \leq p\}, T_U, F_U)$, where $T_U = \{u_1, \ldots, u_p\}$ and $(x, y) \in F_u \iff \exists k : 1 \leq k \leq p \wedge ((x, y) = (\mathsf{request_k}, u_k) \vee (x, y) = (u_k, \mathsf{reply_k}))$. All request- and reply-places are removed from the interface by this construction, only the entry-, the exit- and all in msg- and out msg-places are visible at the interface-level.

If we want to emphasize that we are speaking about protocol P and not about P^U, we denote P explicitly as P^0. Only the inner component P^0 has request msg- and reply msg-places visible at the interface.

The component P^U describes the commitments, whereas the environment model U describes the assumptions of P^0. Correctness of an EPPN could only be established in combination with the environment, so the component P^U seems to be – at least preliminarily – the right candidate.

Definition 1 *A protocol P is* well formed w.r.t. the environment U, *iff the protocol P^U with integrated environment U fulfills:*

$$P^U :: \mathsf{entry} + \sum_I \mathsf{in_i} \triangleright \mathsf{exit} + \sum_J \mathsf{out_j}$$

4.4 Correctness in terms of A/C

EPPN have to be considered in combination with a well behaving environment. Up to now this fact has to be reflected by analyzing the protocol P in combination with an explicitly given environment. Here, we have the simplest environment U (resulting in the extension P^U).

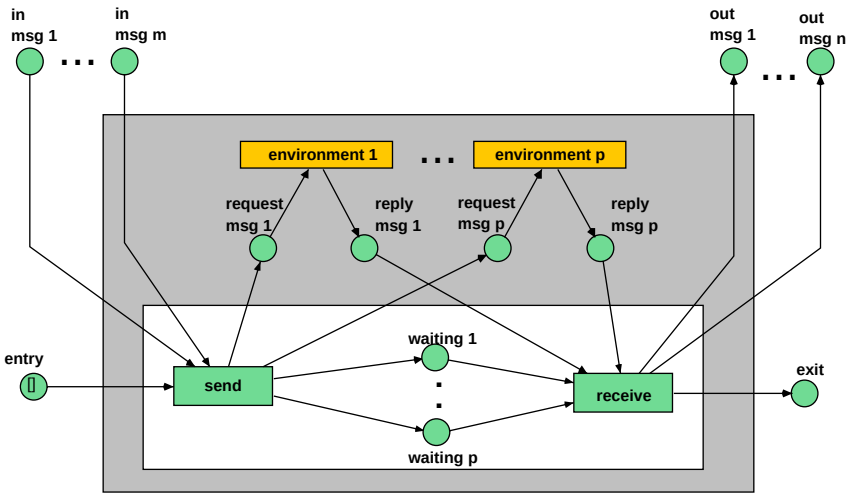


Figure 13: Protocol P^U with integrated environment assumptions

As we have pointed out earlier, this approach is undesirable, since the environment is unknown in advance. As we have stated also, the modeling of the assumptions towards the environment rather than the environment itself plays a central role in the description of agent systems. Therefore, the notion of assumptions and commitments is used for describing the correctness of protocols. A protocol P is defined by the assumption, that all request messages are finally replied by the environment. Under this assumption the commitment, that all out messages are produced on termination, could be fulfilled.

Theorem 1 *An EPPN P with environment U is* well formed w.r.t. *U, iff it fulfills*

$$P :: \mathsf{entry} + \sum_I \mathsf{in_i} \triangleright \sum_K \mathsf{wait_k} + \sum_K \mathsf{req_k}$$

and

$$P :: \sum_K \mathsf{wait_k} + \sum_K \mathsf{repl_k} \triangleright \mathsf{exit} + \sum_J \mathsf{out_j}.$$

In this case the environment assumption*:*

$$U :: \sum_K \mathsf{req_k} \triangleright \sum_K \mathsf{repl_k}$$

establishes the commitment *of the protocol:*

$$P :: \mathsf{entry} + \sum_I \mathsf{in_i} \triangleright \mathsf{exit} + \sum_J \mathsf{out_j}$$

4.5 Explicit notion of assumptions and commitments

EPPN could be formally described and analyzed in terms of the A/C notion. This approach should be enhanced by one step: A/C notions should not only establish the possibility for modular correctness proofs – it should further be an integral part of the Petri net model. Assumptions towards the environment are integrated in the model of the protocol P. This approach enables the modeler to describe directly the scenario assumed. Petri net protocols allowing this style of modeling are called A/C-protocols.

In the MULAN framework A/C-protocols are based on A/C-Petri nets. A/C-Petri nets are described – as seen before – in terms of synchronous channels, more precisely by up-links: The notion of an up-link describes the "passive" part of two synchronizing transitions. This is exactly

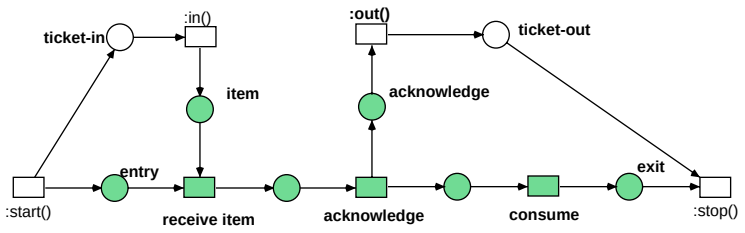


Figure 14: The consumer protocol

what we want for a model of the environment: an external action matching an assumption about an external action.

So, A/C-protocols, as the model of EPPN in the framework of MULAN, contain additional environment transitions. We pick up our introducing example of the producer-consumer scenario. The protocols nets in figure 4 and 5 are now described by the A/C approach (cf. figure 14 and 15).[19] As one can see, A/C-protocols in MULAN describe the assumptions made by the producer in an explicit way. The model is as expressive as the model P^U in figure 13, without the disadvantage of including directly the environment U. This is done by the assumption part of the protocol, depicted by unfilled net elements: the transitions with the inscription (:in() and :out() as well as :start() and :stop()). Additionally, the places info, $\text{ticket}_{\text{in}}$, and $\text{ticket}_{\text{out}}$ are declared to be part of the assumptions.[20] The place info contains data that is needed to match each request and reply.

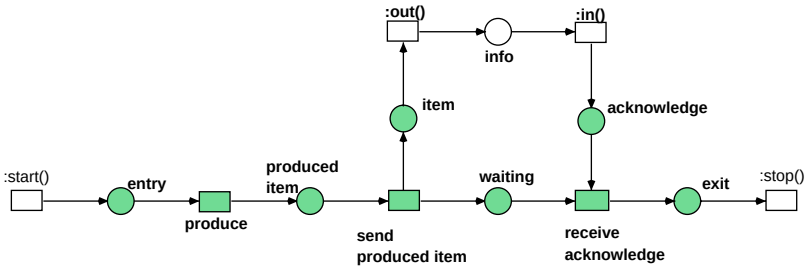


Figure 15: The producer protocol

How this assumption part could be exploited for verification is shown in the exemplary correctness proof of the producer-consumer scenario.

5 Example: The producer-consumer scenario

The producer-consumer scenario is described by a conversation of two agents. One agent uses the consumer protocol C (figure 14), the other the producer protocol P (figure 15). To express the A/C style used for both protocols the consumer protocols is denoted as $C^{A/C}$ and the producer as $P^{A/C}$. The assumptions that are made by each agent towards the other could easily be identified, since the elementary functionality of the protocols (P^0 and C^0) are depicted by filled elements, while the assumption (P^A and C^A) are depicted by unfilled ones.

The consumer protocol is well formed, so we have for C^0

$$C^0 :: \mathsf{entry} + \mathsf{item} \triangleright \mathsf{exit} + \mathsf{acknowledge}$$

[19] Here, we abstract from the token color. This can be done, since the color is only used for documentation purposes and does not contain an inner structure.

[20] Note, that places in A/C Petri nets are not subdivided like transitions into normal and environmental ones, so this fact is only a point of presentation.

and for the extension $C^{A/C}$ with an integrated environment model

$$\begin{array}{rcl} C^{A/C} & :: & \mathsf{entry} + \mathsf{item} \triangleright \mathsf{exit} + \mathsf{acknowledge} \\ (C^{A/C}.e?{:}\mathsf{start}\,!{-}) & : & 0 \triangleright \mathsf{ticket_{in}} + \mathsf{entry} \\ (C^{A/C}.x?{:}\mathsf{stop}\,!{-}) & : & \mathsf{ticket_{out}} + \mathsf{exit} \triangleright 0 \\ (C^{A/C}.c_1?{:}\mathsf{in}\,!{-}) & : & \mathsf{ticket_{in}} \triangleright \mathsf{item} \\ (C^{A/C}.c_2?{:}\mathsf{out}\,!{-}) & : & \mathsf{acknowledge} \triangleright \mathsf{ticket_{out}} \end{array}$$

The producer P ist based on the request-reply interaction. Obviously P^0 describes a well formed protocol:

$$\begin{array}{rcl} P^0 & :: & \mathsf{entry} \triangleright \mathsf{waiting} + \mathsf{item} \\ P^0 & :: & \mathsf{waiting} + \mathsf{acknowledge} \triangleright \mathsf{exit} \end{array}$$

The producer protocol $P^{A/C}$ with integrated environment fulfills:

$$\begin{array}{rcl} P^{A/C} & :: & \mathsf{entry} \triangleright \mathsf{waiting} + \mathsf{item} \\ P^{A/C} & :: & \mathsf{waiting} + \mathsf{acknowledge} \triangleright \mathsf{exit} \\ (P^{A/C}.e?{:}\mathsf{start}\,!{-}) & : & 0 \triangleright \mathsf{entry} \\ (P^{A/C}.x?{:}\mathsf{stop}\,!{-}) & : & \mathsf{exit} \triangleright 0 \\ (P^{A/C}.c_1?{:}\mathsf{out}\,!{-}) & : & \mathsf{item} \triangleright \mathsf{info} \\ (P^{A/C}.c_2?{:}\mathsf{in}\,!{-}) & : & \mathsf{info} \triangleright \mathsf{acknowledge} \end{array}$$

Under the assumption

$$P^A :: \mathsf{item} \triangleright \mathsf{acknowledge}$$

the producer protocol fulfills the commitment

$$P^{A/C} :: \mathsf{entry} \triangleright \mathsf{exit}$$

This is proven by the following derivation

$$\begin{array}{lll} P^{A/C} :: \mathsf{entry} & & \text{Producer } P \\ \triangleright \quad \mathsf{waiting} + \mathsf{item} & & \text{Assumption } P^A \\ \triangleright \quad \mathsf{waiting} + \mathsf{acknowledge} & & \text{Producer } P \\ \triangleright \quad \mathsf{exit} & & \end{array}$$

As one can see, the assumption/commitment proof style is strongly related to the analysis of the embedded nets: $\lceil P^{A/C} \rceil$. In a suitable environment, the assumption part P^A of the embedded A/C protocol is activated and therefore fulfilled:

$$\lceil (P^{A/C}.c_1?{:}\mathsf{out}\,!{-}) \rceil = (P^{A/C}.c_1?{-}!{-}) \qquad \lceil P^{A/C}.c_2?{:}\mathsf{out}\,!{-}) \rceil = (P^{A/C}.c_2?{-}!{-})$$

So, we have:

$$\begin{array}{rcl} \lceil P^{A/C} \rceil & :: & \mathsf{item} \triangleright \mathsf{info} \\ \lceil P^{A/C} \rceil & :: & \mathsf{info} \triangleright \mathsf{acknowledge} \\ \hline \lceil P^{A/C} \rceil & :: & \mathsf{item} \triangleright \mathsf{acknowledge} \end{array}$$

A conversation is modeled by the composition of P and C via communication channels[21]. In our example they are named $c_!$ (the channel for sending the item) and $c_?$ (the channel for receiving the acknowledge). The conversation is denoted by $P||_{\{c_!,c_?\}}C$. In this composition the consumer's commitments fulfill the producer's assumption, the composition discharges the assumptions by commitments. This can be seen from the following derivation: Assume, that channel $c_!$ links the up-links $C.c_1$ and $P.c_1$:

$$c_! = (C.c_1 \sim P.c_1) : \mathsf{C.ticket_{in}} + \mathsf{P.item} \triangleright \mathsf{C.item} + \mathsf{P.info}$$

[21] Do not confuse communication channels of the conversation with the synchronous channels of Renew.

Assume, that channel $c_?$ links the up-links $C.c_2$ and $P.c_2$:

$$c_? = (C.c_2 \sim P.c_2) : \mathsf{C.acknowledge} + \mathsf{P.info} \triangleright \mathsf{C.ticket_{out}} + \mathsf{P.acknowledge}$$

Since all needed formulas of P^0 and C^0 are obviously stable under embedding they hold also – by the embedding rule – for $P^{A/C}||_{\{c_!,c_?\}}C^{A/C}$. With use of the substitution rule for $\triangleright$ it follows:

$$\begin{array}{lll} P^{A/C}||_{\{c_!,c_?\}}C^{A/C} :: & \mathsf{C.entry} + \mathsf{C.ticket_{in}} + \mathsf{P.entry} & \text{Producer} \\ & \triangleright\ \mathsf{C.entry} + \mathsf{C.ticket_{in}} + \mathsf{P.item} + \mathsf{P.waiting} & c_! \\ & \triangleright\ \mathsf{C.entry} + \mathsf{C.item} + \mathsf{P.info} + \mathsf{P.waiting} & \text{Consumer} \\ & \triangleright\ \mathsf{C.exit} + \mathsf{C.acknowledge} + \mathsf{P.info} + \mathsf{P.waiting} & c_? \\ & \triangleright\ \mathsf{C.exit} + \mathsf{C.ticket_{out}} + \mathsf{P.acknowledge} + \mathsf{P.waiting} & \text{Producer} \\ & \triangleright\ \mathsf{C.exit} + \mathsf{C.ticket_{out}} + \mathsf{P.exit} & \end{array}$$

The conversation formed by the two protocols fits with respect to their environment assumptions: $C.Commitment \Longrightarrow P.Assumption$. In general, protocols, which join a conversation, are implicitly ordered by such an implication. Such an order is called *conversation order* – a central structuring element for well formed conversations; well formed conversations and acyclic A/C proofs are therefore strongly related. Due to the limited space we cannot discuss the details here any further.

6 Conclusion

This presentation gives insight to our unifying approach for multi agent systems, realized in the MULAN-architecture .

The general architecture is based on high level Petri nets, namely reference nets. The formalism of reference nets is grounded on the "nets within nets" paradigm formulated by Valk. This paradigm fits well, as we have shown both with respect to modeling and verificational aspects, in the context of agent systems, since it can naturally express the key concepts of agent systems, like mobility, adaptation, and cooperation. The paradigm also meets well with the requirements of verification, like concurrency, modularity, and explicit notion of environment assumptions.

Multi agent systems are based on the concepts of mobility, cooperation, and adaptation. In this work, we have focused on one central aspect of cooperation which is described by agents conversation. We have shown that conversations could naturally be modeled and verified within our approach of A/C Petri net protocols.

Due to size limitations of this presentation, only the key concepts could be demonstrated; we have chosen the well known producer-consumer scenario to describe integrated A/C modeling and verification. Future publication will show in detail how the verification approach, proposed by MULAN, scales up with complex conversations and how the remaining concepts – mobility and adaptation – are integrated into the formalism in detail.

References

[AFdR80] K. R. Apt, N. Francez, and W.P. de Roever. A proof system for communicating sequential processes. *ACM TOPLAS*, 2(3):359–385, 1980.

[BBC+99] B. Barras, S. Boutin, C. Cornes, J. Courant, Y. Coscoy, D. Delahaye, D. de Rauglaudre, J.-Ch. Fillitre, E. Giménez, H. Herbelin, G. Huet, H. Laulhère, C. Muñoz, C. Murthy, C. Parent-Vigouroux, P. Loiseleur, Ch. Paulin-Mohring, A. Sabi, and B. Werner. *The Coq Proof Assistent – Reference Manual, Version 6.3.1, Coq Project*, December 1999. `http://pauillac.inria.fr/coq`.

[BDM+99] M. Bozzano, G. Delzanno, M. Martelli, V. Mascardi, and F. Zini. Logic Programming & Multi-Agent Systems: a Synergic Combination for Applications and Semantics. In K.R. Apt, V.W. Marek, M. Truszczynski, and D.S. Warren, editors, *The Logic Programming Paradigm: a 25-Year Perspective*, pages 5–32. Springer Verlag, 1999.

[Ber87] G. Berthelot. Transformations and decompositions of nets. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri nets: Central models and their properties*, number 254/255 in LNCS. Springer-Verlag, 1987.

[BOP00] Bernhard Bauer, James Odell, and H. van Dyke Parunak. Extending UML for Agents. In *Proceeding of Agent-Oriented Information Systems Workshop*, pages 3 – 17, 2000.

[BT87] E. Best and P.S. Thiagarajan. Some classes of live and safe Petri nets. In Klaus Voss, editor, *Concurrency and nets*, pages 71–94, Berlin, Germany, 1987. Gesellschaft für Mathematik und Datenverarbeitung, Springer-Verlag.

[CCF+99] R. Scott Cost, Ye Chen, T. Finin, Y. Labrou, and Y. Peng. Modeling agent conversation with colored Petri nets. In *Working notes on the workshop on specifying and implementing conversation policies (Autonomous agents '99)*, 1999.

[CFL+98] R.S. Cost, T Finin, Y. Labrou, X. Luan, Y. Peng, L. Soboroff, J. Mayfield, and A. Voughannanm. Jackal: A Java-based Tool for Agent Development. In *Working Notes of the Workshop on Tools for Developing Agents, AAAI'98*, pages 73–82. AAAI Press, 1998.

[CGG99] Luca Cardelli, Andrew D. Gordon, and G. Ghelli. Mobility types for mobile ambients. In *Proc. of the ICALP'99*, LNCS 1644, pages 230–239. Springer-Verlag, 1999.

[CH94] S. Christensen and N.D. Hansen. Coloured Petri nets extended with channels for synchronous communication. In Rober Valette, editor, *Application and Theory of Petri Nets 1994, Proc. of 15th Intern. Conf. Zaragoza, Spain, June 1994*, LNCS, pages 159–178, June 1994.

[DE96] A. Diagne and Pascal Estraillier. Formal specification and design of distributed systems. In *Proceedings of the Formal Methods for Object-based Open Distribured Systems*, Paris, France, 1996. IFIP.

[EC82] E. A. Emerson and E. M. Clarke. Using branching time temporal logics to synthesize synchronisation sceletons. *Sci. Comput. Program.*, 2:241–266, 1982.

[Far00] B. Farwer. *Linear Logic Based Calculi for Object Petri Nets.* Logos Verlag, Berlin, 2000.

[FIP] Foundation for Intelligent Physical Agents. `http://www.fipa.org`.

[FIP98a] FIPA. FIPA 97 Specification, Part 1 - Agent Management. Technical report, Foundation for Intelligent Physical Agents, `http://www.fipa.org`, Oktober 1998.

[FIP98b] FIPA. FIPA 97 Specification, Part 2 - Agent Communication Language. Technical report, Foundation for Intelligent Physical Agents, `http://www.fipa.org`, Oktober 1998.

[FL97] Tim Finin and Yannis Labrou. A Proposal for a new KQML Specification. Technical report, University of Maryland, Februar 1997.

[GPdFC98] Gustavo M. Gois, Angelo Perkusich, Jorge C. A. de Figueiredo, and Evandro B. Costa. Towards a multi-agent interactive learning environment oriented to the Petri net domain. In *Proc. IEEE Int. Conf. on Systems, Man, and Cybernetics (SMC'98), 11-14 October 1998, San Diego, USA*, pages 250–255, October 1998.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communication of the ACM*, 12:576–580, 1969.

[HR00] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and reasoning about systems.* Cambridge University Press, 2000.

[Jen92] K. Jensen. *Coloured Petri nets, Basic Methods, Analysis Methods and Practical Use*, volume 1 of *EATCS monographs on theoretical computer science.* Springer-Verlag, 1992.

[Jen00] Nicholas R. Jennings. On agent-based software engineering. *Artificial Intelligence*, 117:277–296, 2000.

[KMR01] Michael Köhler, Daniel Moldt, and Heiko Rölke. Modeling the behaviour of Petri net agents. In J. M. Colom and M. Koutny, editors, *Proceedings of the 22st Conference on Application and Theory of Petri Nets*, volume 2075 of *LNCS*, pages 224–241. Springer-Verlag, June 2001.

[Köh00a] Michael Köhler. Branching process of Petri nets - an unifying approach. Technical Report 293/00, University of Hamburg, Department for Computer Science, Vogt-Kölln Str. 30, 22527 Hamburg, Germany, 2000.

[Köh00b] Michael Köhler. Distribution references and undecided markings. Technical Report 292/00, University of Hamburg, Department for Computer Science, Vogt-Kölln Str. 30, 22527 Hamburg, Germany, 2000.

[Köh01] Michael Köhler. Algebraische Darstellung von Objektnetzsystemen. To be published as a Technical Report, University of Hamburg, Department for Computer Science, Vogt-Kölln Str. 30, 22527 Hamburg, Germany, 2001.

[Kum98] Olaf Kummer. Simulating synchronous channels and net instances. In J. Desel, P. Kemper, E. Kindler, and A. Oberweis, editors, *Forschungsbericht Nr. 694: 5. Workshop Algorithmen und Werkzeuge für Petrinetze*, pages 73–78. Universität Dortmund, Fachbereich Informatik, 1998.

[KW98] Olaf Kummer and Frank Wienberg. *Reference net workshop (Renew).* Universität Hamburg, http://www.renew.de, 1998.

[MC81] J. Misra and M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(4):417–426, 1981.

[MM90] J. Meseguer and U. Montanari. Petri nets are monoids. *Information and Computation*, 88(2):105–155, October 1990.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts 1-2. *Informartion and computation*, 100(1):1–77, 1992.

[OG76] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta informatica*, 6:319–340, 1976.

[Rei85] Wolfgang Reisig. *Petri Nets: An Introduction.* Springer-Verlag, 1985.

[RJB99] J. Rumbaugh, I. Jacobson, and G. Booch. *The unified modeling language reference manual: The definitive reference to the UML from the original designers.* Addison-Wesley object technology series. Addison-Wesley, Reading, Mass., 1999.

[SHM99] A. El Fallah Seghrouchni, S. Haddad, and H. Mazouzi. A formal study of interactions in multi-agent systems. In *14th ISCA-CATA*, Cancun, Mexique, April 1999.

[Val87] Rüdiger Valk. Modelling of task flow in systems of functional units. Technical Report FBI-HH-B-124/87, Universität Hamburg, 1987.

[Val96] Rüdiger Valk. Concurrency in communicating object Petri nets. In G. Agha, F. de Cindio, and G. Rozenberg, editors, *Advances in Petri Nets*, volume 32 of *Lecture Notes in Computer Science*, 1996.

[Val98] Rüdiger Valk. Petri nets as token objects: An introduction to elementary object nets. In Jörg Desel and Manuel Silva, editors, *Application and Theory of Petri Nets*, volume 1420 of *LNCS*, pages 1–25, June 1998.

[VC99] Jan Vitek and Giuseppe Castagna. Seal: A framework for secure mobile computations. *Internet Programming Languages*, 1999.

[Vig98] G. Vigna, editor. *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science.* Springer-Verlag, 1998.

[Wei99] G. Weiss, editor. *Multiagent systems.* MIT Press, 1999.

[Woo99] Michael Wooldridge. Intelligent agents. In Weiss [Wei99], chapter 1.

[Zwi89] J. Zwiers. *Compositionality, concurrency, and partial correctness: proof theories for networks of processes and their relationship.* LNCS 321. Springer-Verlag, 1989.

Modeling State-Dependent Objects using Colored Petri Nets

Robert G. Pettit IV[1] and Hassan Gomaa[2]

[1]The Aerospace Corporation, 15049 Conference Center Drive,
Chantilly, Virginia, USA 20151
rob.pettit@aero.org

[2]George Mason University, Department of Information and Software Engineering,
Fairfax, Virginia, USA 22030-4444
hgomaa@gmu.edu

Abstract. This paper describes an approach for using Petri nets to model and analyze the behavioral characteristics of state-dependent objects represented in the Unified Modeling Language (UML). Specifically, this paper describes an approach for systematically mapping UML state-dependent objects and their corresponding statecharts into colored Petri nets. This work is part of an on-going effort to automate the behavioral analysis of concurrent and real-time object-oriented software designs. The benefit of this approach is that by providing a systematic means for modeling state-dependent objects using colored Petri nets and relating this to the larger research effort, the overall concurrent architecture may then be modeled and analyzed using a single, cohesive technique.

Keywords: UML; Statechart; Colored Petri Net; Behavioral Analysis; COMET

1 Introduction

This paper presents an approach for using colored Petri nets to model and subsequently validate the behavioral characteristics of state-dependent objects represented in the Unified Modeling Language (UML) [1;2]. This research represents part of a larger effort [3] to integrate colored Petri nets with UML software architectures created using the COMET method [4]. The goal of this overall effort is to provide a systematic and seamless integration of CPNs with object-oriented software architectures in order to effectively model and analyze the behavioral properties of an architecture prior to implementation.

In COMET (Concurrent Object Modeling and Architectural Design Method), state-dependent objects are active (asynchronous) objects that react to stimuli based on the current state of the object. Each state-dependent object has an associated statechart that defines the state-based behavior for that object. Typically, a state-dependent object will have one input interface (operation) for processing events. When an event is received, the state-dependent object will then use the encapsulated statechart information to determine the appropriate behavior to execute. Based on the input that was received and the current state of the object, this behavior could range from simply changing states; executing some action; sending a message to other objects; or no behavior at all.

There are several tools currently available to simulate statechart execution. The benefit of this approach is that by providing a systematic means for modeling state-dependent objects using colored Petri nets and relating this to the larger research effort mentioned above, the overall concurrent architecture may then be modeled and analyzed using a single, cohesive technique.

The Petri net formalism was chosen based on its modeling and analytical power for concurrent systems. There are three general characteristics of Petri nets that make them

interesting in capturing concurrent, object-oriented behavioral specifications. First, Petri nets allow the modeling of concurrency, synchronization, and resource sharing behavior of a system. Second, there are many theoretical results associated with Petri nets for the analysis of such issues as deadlock detection and performance analysis. Finally, the integration of Petri nets with an object-oriented software architecture provides a means for automating behavioral analysis.

The basic notation for Petri nets is a bipartite graph consisting of places and transitions that alternate on a path and are connected by directional arcs [5]. In general, circles represent places, whereas bars or boxes represent transitions. Tokens are used to mark places, and under certain enabling conditions, transitions are allowed to *fire*, thus causing a change in the placement of tokens.

A colored Petri net (CPN) is a special case of Petri net in which the tokens have identifying attributes; in this case the color of the token [6]. At first, colored Petri nets seem less intuitive than the basic Petri net. However, by allowing the tokens to have an associated attribute, colored Petri nets scale to large problems much better than basic Petri nets.

2 The COMET Method

COMET is a Concurrent Object Modeling and Architectural Design Method for the development of concurrent applications, in particular distributed and real-time applications [4]. As the UML is now the standardized notation for describing object-oriented models [1,2], the COMET method uses the UML notation throughout.

The COMET Object-Oriented Software Life Cycle is highly iterative. In the Requirements Modeling phase, a use case model is developed in which the functional requirements of the system are defined in terms of actors and use cases.

In the Analysis Modeling phase, static and dynamic models of the system are developed. The static model defines the structural relationships among problem domain classes. Object structuring criteria are used to determine the objects to be considered for the analysis model. A dynamic model is then developed in which the use cases from the requirements model are refined to show the objects that participate in each use case and how they interact with each other. In the dynamic model, state dependent objects are defined using statecharts.

In the Design Modeling phase, an Architectural Design Model is developed. Subsystem structuring criteria are provided to design the overall software architecture. For distributed applications, a component based development approach is taken, in which each subsystem is designed as a distributed self-contained component. The emphasis is on the division of responsibility between clients and servers, including issues concerning the centralization vs. distribution of data and control, and the design of message communication interfaces, including synchronous, asynchronous, brokered, and group communication. Each concurrent subsystem is then designed, in terms of active objects (tasks) and passive objects. Task communication and synchronization interfaces are defined. The performance of real-time designs is estimated using an approach based on rate monotonic analysis.

The aspect of the COMET method that is specifically addressed in this paper is the emphasis on dynamic modeling, in the form of both object interaction modeling and finite state machine modeling, describing in detail how object collaborations and statecharts [8,9,10] work together.

3 Statecharts in the COMET Method

3.1 State Transition Diagrams and Statecharts

Traditionally, finite state machines were modeled by means of state transition diagrams or state transition tables. One of the potential problems of state transition diagrams is the proliferation of states and transitions, thereby making the state transition diagram very cluttered and difficult to read. A very important way of simplifying state transition diagrams was made by Harel with the introduction of statecharts [8,10], which increases the modeling power of state transition diagrams by introducing superstates and the hierarchical decomposition of superstates.

Significant simplification of statecharts can often be achieved through the use of hierarchical decomposition of states, where a superstate is decomposed into two or more interconnected substates. This decomposition is sometimes referred to as the **or** decomposition, because being in the superstate means that the statechart is in one and only one of the substates. The hierarchical statechart notation also allows a transition out of every one of the substates on a statechart to be aggregated into a transition out of the superstate. Careful use of this feature can significantly reduce the number of state transitions on a statechart.

Another kind of hierarchical state decomposition supported is the **and** decomposition. That is, a state on one statechart can be decomposed into two or more concurrent statecharts. When the higher-level statechart is in the superstate, it is simultaneously in one of the substates on the first lower-level concurrent statechart *and* in one of the substates on the second lower-level concurrent statechart.

Although the name concurrent statechart implies that there is concurrent activity within the object containing the statechart, the **and** decomposition can be used to show different aspects of the same object, which are not concurrent. This latter approach is used in the COMET method.

3.2 State Dependent Objects and Statecharts

State dependent objects are control objects whose behavior depends not only on the input received, but also on the current state. In COMET, a state dependent object is described by means of a statechart. A complex system can have many state dependent objects and hence several statecharts. COMET encourages the design of objects with only one thread of control; to address concurrency, multiple concurrent objects can be used. The Cruise Control system described in this paper has one state dependent object and hence one statechart. However, in other case studies given in [4], there are examples of distributed control in which the control aspects of the system are distributed among many state dependent objects, each described by its own statechart. If there are many objects of the same type, then each object will execute an instance of the same statechart.

4 Modeling UML State-Dependent Objects Using Colored Petri Nets

Using the COMET method, state-dependent objects are defined to encapsulate the behavior specified by a statechart. These state-dependent objects provide an interface for receiving events and then perform some behavior based on the input event, the current state, and the state-transition specifications of the encapsulated statechart.

To fit within the context of modeling a large-scale concurrent software architecture, the approach used in this paper to specifically model state-dependent objects must address both the high-level behavioral structure of state-dependent objects as well as the specific state-dependent behavior of the associated statechart. To capture both of these aspects, the resulting CPN model is structured using a series of hierarchical decompositions as described in the following sections.

4.1 CPN Structural Model of State-Dependent Objects

Using our approach, the top-level CPN model captures the high-level state-dependent object as a whole. This top-level model is illustrated in Figure 1. At this level, we can see the interface between the state-dependent object and its surrounding environment, including input events, output actions, the current state, and the high-level control flow within the object.

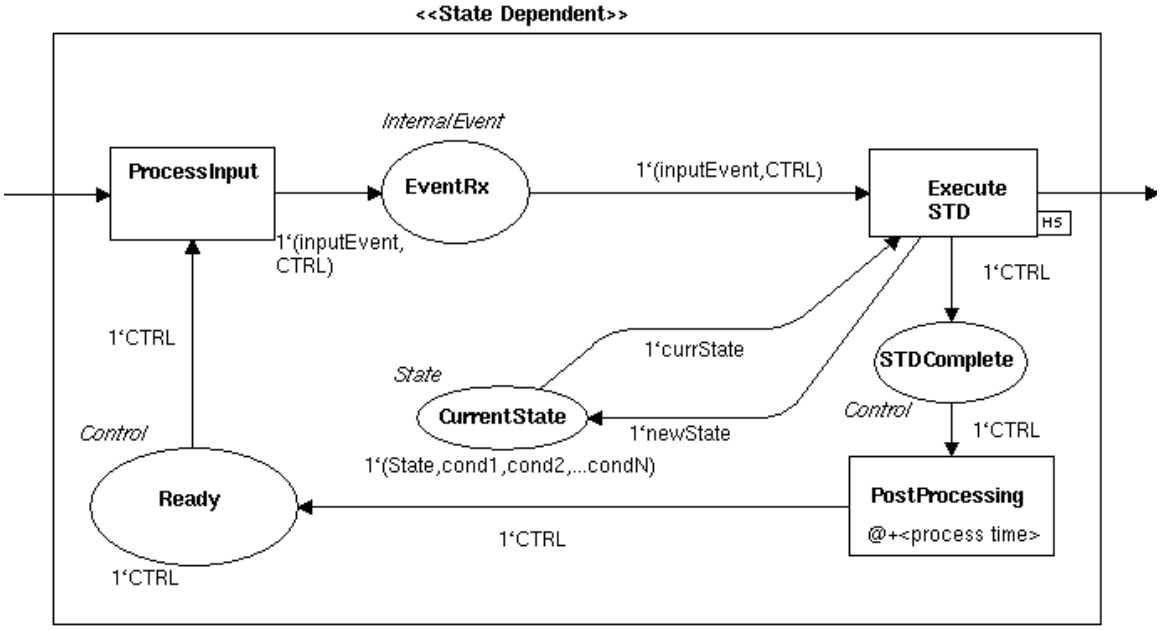


Figure 1. Top-Level CPN Model for State-Dependent Objects

In the COMET method, state-dependent objects are modeled as *active* objects in UML, thus indicating that they operate concurrently with their own thread of control. In our CPN model, this concept is maintained by using a control token (labeled *CTRL* in the CPN diagrams) to model the flow of control within the object. Each active object is modeled with its own control token, which, in addition to modeling the flow of control, is used to model synchronization with other objects and (with a timestamp) is used to simulate the progression of time for each object.

A state-dependent object modeled with the CPN segment from Figure 1 would begin its lifecycle with a control token residing in the *Ready* place. Thus, the *ProcessInput* transition would then be enabled by the presence of an input event. Upon firing, *ProcessInput* would pair the input event with the control token and pass this tuple to the *EventRx* place, indicating that an event has been received and needs to be processed through the state transition diagram. The *ExecuteSTD* transition is decomposed to model the specific statechart behavior encapsulated by the state-dependent object. This decomposition is described in Section 4.2.

Once the statechart model has completed its execution (including any output actions), the control token is passed to the *STDComplete* place. The *PostProcessing* transition then increments the time tag on the control token (to simulate the processing time for the state-dependent object to handle an event) and returns the control token to the *Ready* place to await the next event.

One final note to be aware of at this level is the definition of state. The current state is maintained in the *CurrentState* place. The *ExecuteSTD* transition (and its subsequent decompositions) remove the current state from this place with each input event and return the modified (new) state upon completion. The state of the object itself is actually represented by a tuple stored at *CurrentState*. This tuple not only contains the "state" of the system as we would normally consider it, but it also contains the current status of system conditions used to make transition determinations in the statechart. Use of these conditions combined with the state is further illustrated in Section 5 with an example from the Cruise Control System.

4.2 Top-Level CPN Statechart Model

To begin modeling, the actual state-dependent behavior is first converted to a "flattened" representation so that all state transitions are explicitly captured between leaf states. Any necessary conditions that were captured in the hierarchical statechart (e.g. device status) are then represented as conditions on the state transition and are also captured as part of the conditions in the *CurrentState* tuple. Furthermore, "entry" actions for UML states are mapped to actions on the corresponding entry transition(s) to those states. Similarly, "exit" actions are mapped to actions on the corresponding exit transitions(s). Activities that are persistent within a given state (represented by the UML "Do" keyword) are also mapped to actions on the entry transition(s) as these persistent activities are implemented by separate active objects in the COMET method [4].

Working now from a flattened statechart, the *ExecuteSTD* transition is then decomposed into a lower-level CPN. This first-level decomposition provides one CPN transition for each (leaf) state in the object's statechart. Furthermore, the places providing tokens to and receiving tokens from *ExecuteSTD* are shown at this level as *port* places. A port place is used as a connector between hierarchical levels of CPNs. Thus, the *EventReceived* place shown at the root level providing input to *ExecuteSTD* is identical to all subsequent instances of *EventReceived* in the various levels of *ExecuteSTD's* decompositions.

Figure 2 provides an illustration of this first-level decomposition. Note that all transitions (representing the states of the object's statechart) are connected to the *EventReceived*, *STDComplete*, and *CurrentState* places. Additionally, if a given state generates an external action, appropriate links will also be added to handle those cases. Furthermore, each CPN "*State*" transition in Figure 2 uses an arc inscription on the input arc to enable the transition only if the current state corresponds to the state being modeled by that transition. Thus, the *State1* transition is only enabled when the current state is equal to *State1* and so forth.

4.3 Modeling Behavior for Individual States

To model the specific state-dependent behavior, each of these CPN *State* transitions from Figure 2 is again decomposed into a lower-level CPN segment. This final level of decomposition is shown in Figure 3.

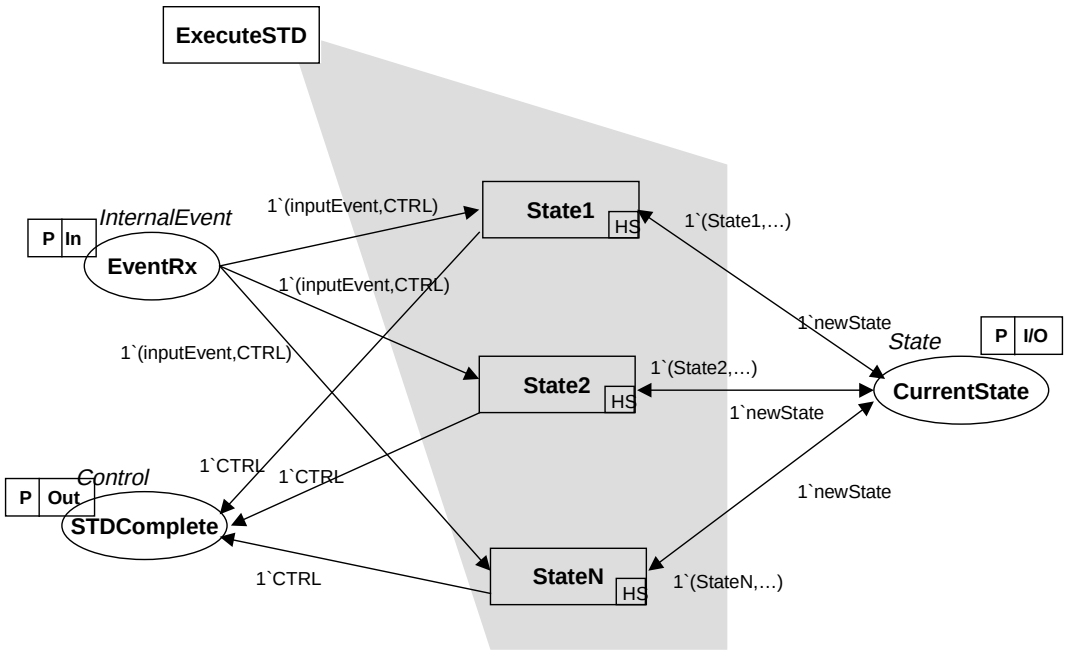


Figure 2. First Level Decomposition of CPN Statechart Model

At this level, there is one transition (*State-n-DetSt*) to determine the next state and the particular action branch that needs to be taken. Once the current state is determined and the appropriate *DetSt* transition is enabled (using the arc inscription from *CurrentState*), this transition executes a code segment to determine the next state (*newState*) and the action branch (*branch*) based on the current state and the input event (*inputEvent*). This code segment simply consists of a case statement in the ML language that specifies the desired output of new state and action branch for each possible input event.

After the *newState* and *branch* values are determined, a tuple containing these values is passed to the *State-n-Branch* place. From this branch place, one branch is chosen based on the value of the *branch* variable. Each branch is a series of alternating transitions and places that model the behavior of the action to be performed. Each transition in a branch is labeled by the name "Action", followed by the branch number (0..n), followed by the step number (a..z). In our approach, branch zero (0) will always be used when no action (other than a possible state change) is required.

For branches with more than one step, an interim place is used to connect each transition in the path. This interim place contains the new state tuple and is labeled by the name, "*State*" followed by the branch number, followed by the previous and next step numbers. For example, in Figure 3, the interim state along branch 1 between steps "a" and "b" is labeled, "*State-n-State1ab*".

Along these action paths, the CPN model may generate tokens to be passed as messages, events, or operation calls to other objects. In the case of asynchronous actions, the branch may be continued uninterrupted. However, in the case of synchronous processing, the next step must wait for the previous step to complete before continuing. This is accomplished by having the next transition enabled by both the interim state place and a return place from the synchronizing action. An example of a synchronous message can be found in the Cruise Control example of Section 5.

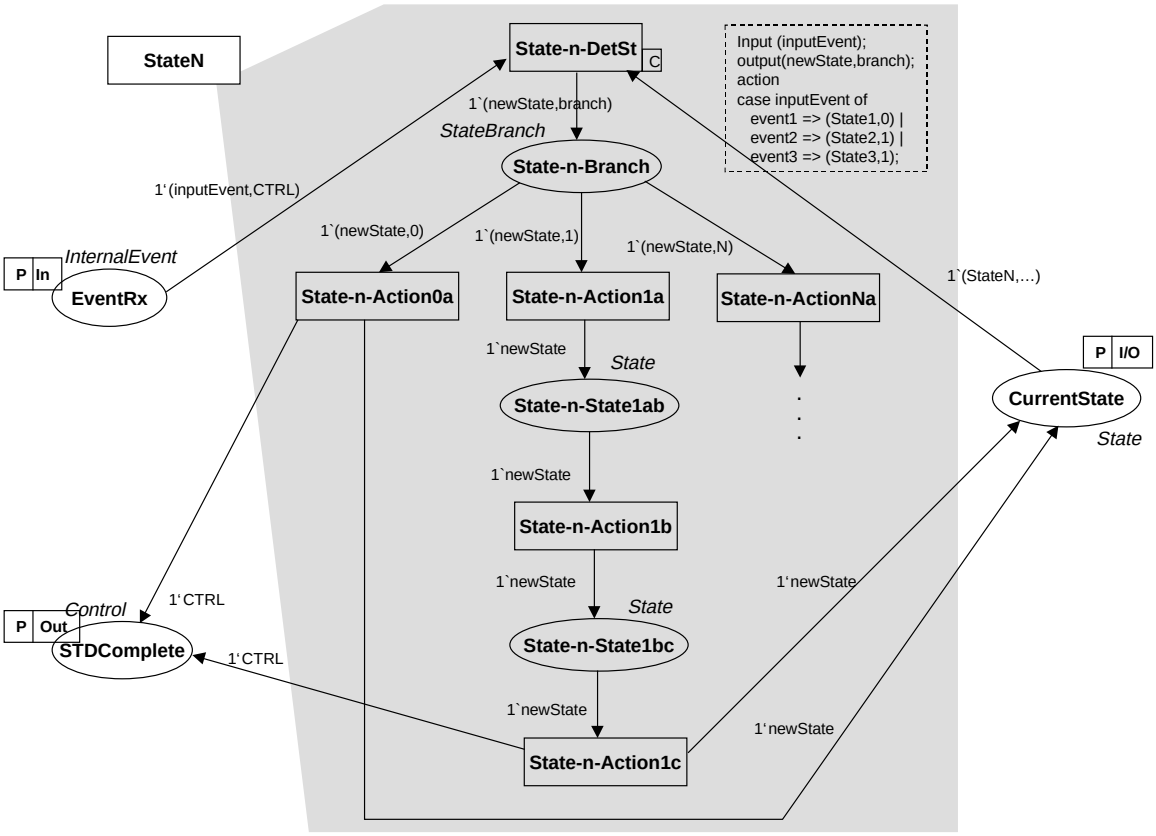


Figure 3. Detailed State Decomposition

Finally, when a given path has completed, it's last action is to update the *CurrentState* place with the new state and to pass a control token to the *STDComplete* place indicating that the current input event has now been processed.

The *STDComplete* place then essentially returns to the top level CPN diagram and passes the control token to the *PostProcessing* transition. This transition performs any necessary post processing actions and uses its time delay to simulate the processing time for the state-dependent object. When this transition completes, the control token is returned to *Ready* place and the model is ready to process the next input event.

At this point, the CPN model for the state-dependent object may be executed through a simulator or analyzed by constructing an occurrence graph of the state-space using tools such as DesignCPN [7]. The state-dependent object may be analyzed in isolation using CPN places for the input and output stubs or it may be integrated into the CPN model of the overall concurrent software architecture.

In the following section, we use an example from the Cruise Control System to illustrate how a specific state-dependent object within a software architecture may be modeled using the approach described in the above sections.

5 Case Study: Cruise Control System

The Cruise Control System [4] is a real-time control system that manages the speed of an automobile based on inputs from the driver (via a lever on the steering column). The behavior of the cruise control is state-dependent in that the executed actions correspond not only to the driver input, but also on the current state of the system and with the status of the engine and the brake.

To illustrate the modeling of state-dependent objects using CPNs, we will use the state-dependent ":*CruiseControl*" object and corresponding statechart from the Cruise Control System. In this example, the :*CruiseControl* object accepts external inputs/events and executes the cruise control statechart. Figure 4 provides a partial collaboration diagram illustrating this state-dependent object along with its interfaces to the rest of the cruise control system. As can be seen from this figure, the *:CruiseControl* object accepts event inputs from the cruise control lever interface indicating whether the driver has selected to accelerate (*Accel*), engaged the cruise (*Cruise*), turned the cruise off (*Cruise Off)*, or requested that cruising be resumed (*Resume*). Furthermore, the *:CruiseControl* object must accept messages from the *SpeedControl* object indicating that the cruising speed has been reached and accept inputs indicating the current status of the brake (*Pressed* or *Released*) and engine (*On* or *Off*). Finally, in terms of output actions, the :*CruiseControl* object must set (or clear) the desired speed and must send appropriate (state-based) commands to the *SpeedControl* object that controls the throttle output and monitors the current speed.

The statechart for the :*CruiseControl* object is shown in Figure 5. The leaf (lowest-level) states for *:CruiseControl* are: Idle, Initial, Accelerating, Cruising, Resuming, and Cruising Off. It is these low-level states that will be used for the CPN state-dependent modeling.

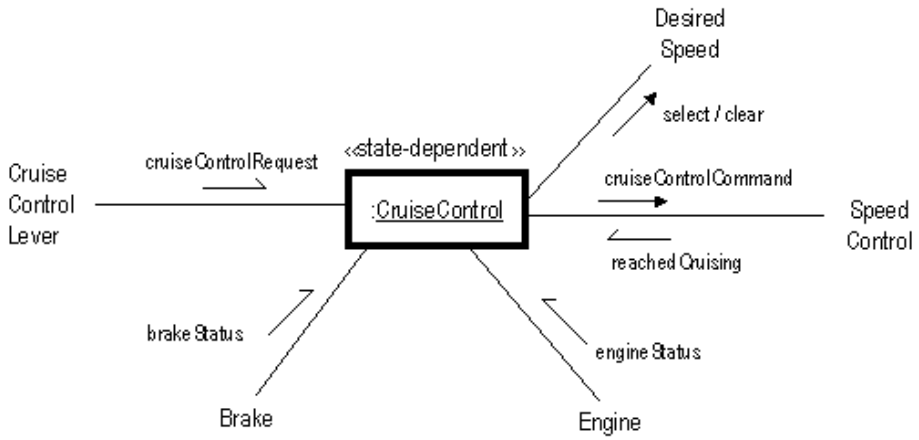


Figure 4. Cruise Control State-Dependent Object

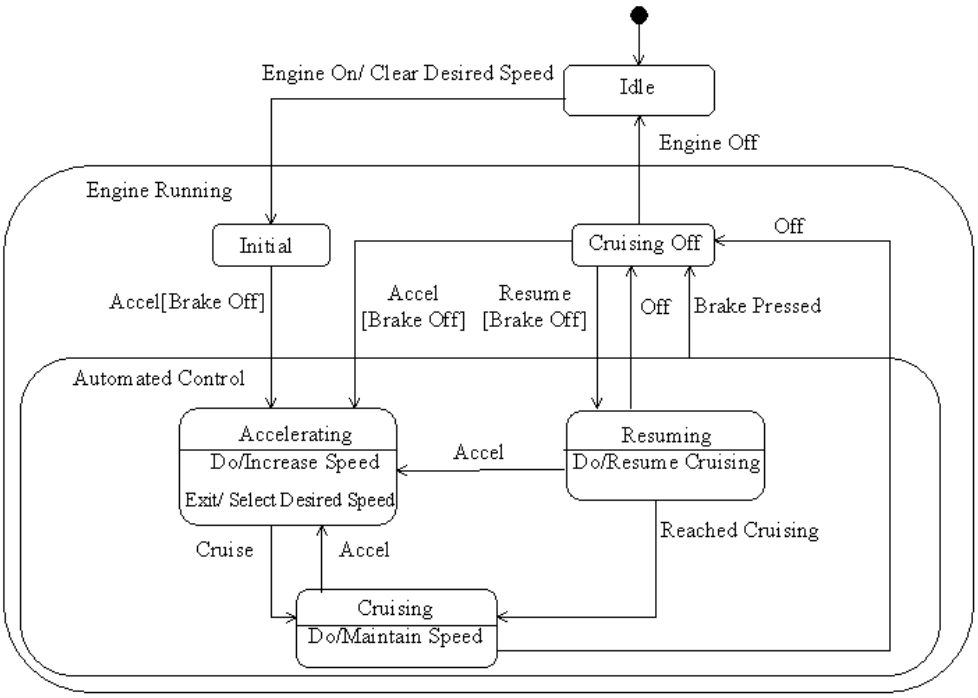


Figure 5. Cruise Control Statechart
Figure extracted from [4] with permission of Addison-Wesley Publishing.

5.1 CPN Model for Cruise Control

To start modeling the behavior of the *:CruiseControl* state-dependent object, we begin by using a CPN segment to capture high-level behavioral properties of the state-dependent object and its connections to the rest of the software architecture. This CPN segment is illustrated in Figure 6.

At this level, the CPN models the general control flow of a state-dependent object. In the case of the *:CruiseControl* object, this segment has been tailored to accept cruise control messages as the input events via the *ProcessInput* transition. Additionally, the *ExecuteSTD* transition is connected to CPN places corresponding to the *Select* and *Clear* operations of the *DesiredSpeed* entity object as well as the places corresponding to the synchronous message buffer for the *SpeedControl* object. The *State* colorset must also be tailored to fit the *:CruiseControl* object and in this case is declared to be a tuple (specifically, a triple) consisting of tokens representing the cruise control leaf states; the engine status; and the brake status.

With the high-level CPN segment in place, we can now begin modeling the state-dependent behavior of *:CruiseControl* by decomposing the *ExecuteSTD* transition into a set of CPN transitions representing the leaf states of the cruise control statechart. This first level decomposition is shown in Figure 7. Using the approach presented in Section 2, one CPN transition is used for each of the six low-level states of the cruise control statechart: Idle, Initial, Accelerating, Cruising, Resuming, and CruisingOff. Each of these CPN transitions accepts a tuple containing the cruise control message and control token from the *EventReceived* place along with the current state from the *CurrentState* place. When tokens are available on both the *EventReceived* and *CurrentState* places, the transitions will then be enabled according to the arc inscription from *CurrentState*. Thus, the *Idle* transition is only enabled if the current state is Idle; the *Initial* transition is only enabled for a current state of Initial; and so forth. Finally, the output from each of these transitions updates the

CurrentState place with the *newState* token and sends a control token to the *STD Complete* place indicating that processing has been completed for the current input event.

The final step in modeling the state-dependent behavior is to decompose each of the transitions in Figure 7 into a CPN segment capturing the behavior for each of the cruise control states. To illustrate this final step, the CPN models for two of the cruise control states (Idle and Accelerating) are described in detail in the following sections. The remaining CPN models can be derived by applying this same systematic approach.

5.1.1 Idle State

The idle state is the starting point for the cruise control statechart. The CPN implementation of the cruise control idle state is shown in Figure 8. In the cruise control statechart shown in Figure 5, there is only one transition identified from the idle state – this occurs when the engine is turned on and the system transitions to the initial state. However, in the CPN implementation of the state-dependent behavior, changes in conditions that affect the cruise control system must also be accounted for. These conditions include changes to the brake and engine status. Thus, the implementation for the idle state not only checks for an "EngineOn" event, but must also check for the other possibilities of "BrakeOn," "BrakeOff," and "EngineOff". When one of these three "events" occurs, the code segment of the *IdleDetSt* transition sets the status variables accordingly, and sets the branch value to 0 (indicating that no further action is necessary). This new tuple is then passed to the *IdleBranch* place. From the *IdleBranch* place, arc inscriptions determine which branch (CPN transition) will be enabled. With the branch value set to 0, the *IdleAction0a* transition is enabled and, when fired, simply updates the *Current State* and sends a control token to the *STD Complete* place indicating that processing is complete for this input event.

The only input event that produces an action and state transition from the idle state is the change in engine status as indicated by a *ccMsg* token with the value, "EngineOn". When this occurs, the code segment of *IdleDetSt* sets the output tuple to be *(Initial,true,bs,1)* indicating that the state should be changed to the initial state; the engine is turned on; the brake status (*bs*) is unchanged; and the action branch is 1. This tuple is then given to the *IdleBranch* place and enables the *IdleAction1a* transition. According to the cruise control statechart, the action that occurs when transitioning from idle to initial states is to clear the desired speed. Thus, the *IdleAction1a* "calls" the clear operation by passing a control token to the *Clear_6* place and then placing the state tuple in the *IdleState1ab* place to continue down this action branch. (The *Clear_6* place is part of the CPN model for the *DesiredSpeed* entity object of the cruise control software architecture. The number six (6) appended to the place name represents the object identifier for *DesiredSpeed*.) Since we are simulating an operation call, we need to have some synchronization so that we wait for the operation to complete before continuing our processing. This is accomplished by having the next transition in the branch, *IdleAction1b*, wait for both a state tuple in the *IdleState1ab* place and a returned control token in the *ClearRtn_6* place (indicating that the clear desired speed operation has completed). Since this is the final step in the action branch, the *IdleAction1b* transition updates the *Current State* place with the new state and conditions and sends a control token to the *STD Complete* place, completing the processing of this input event.

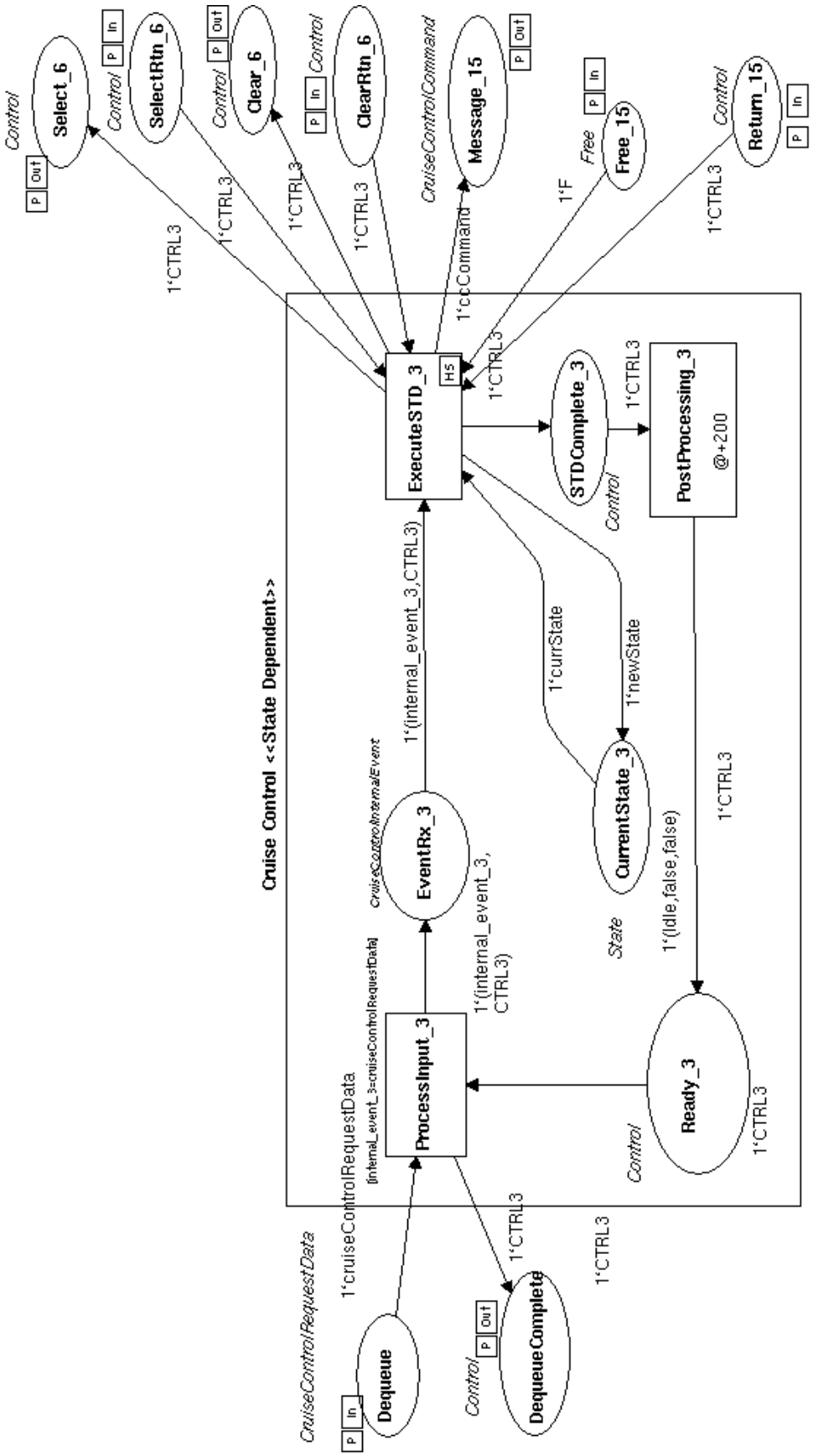


Figure 6. Cruise Control Object: Top-Level CPN Segment

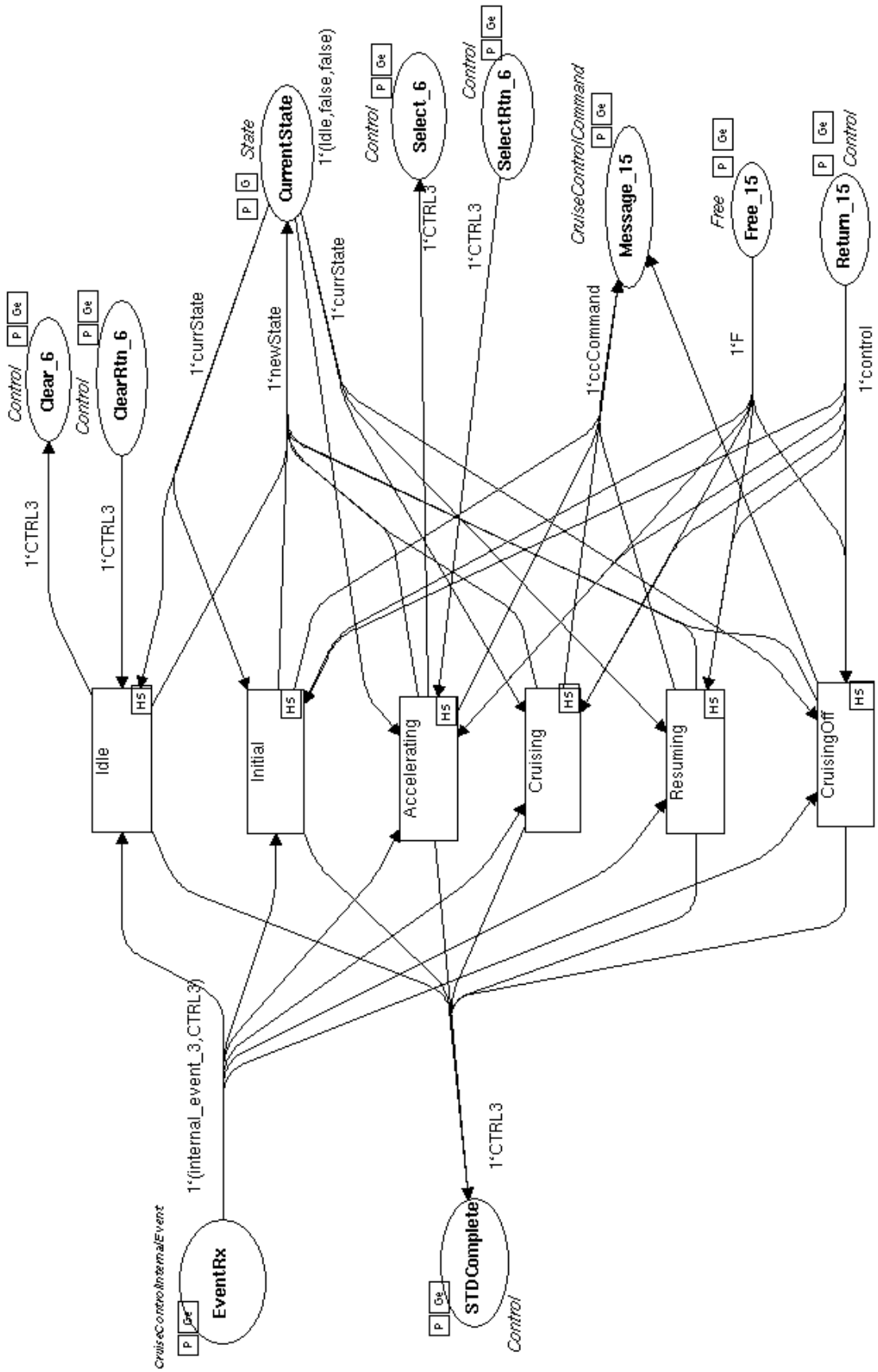


Figure 7. Cruise Control: First Level Decomposition

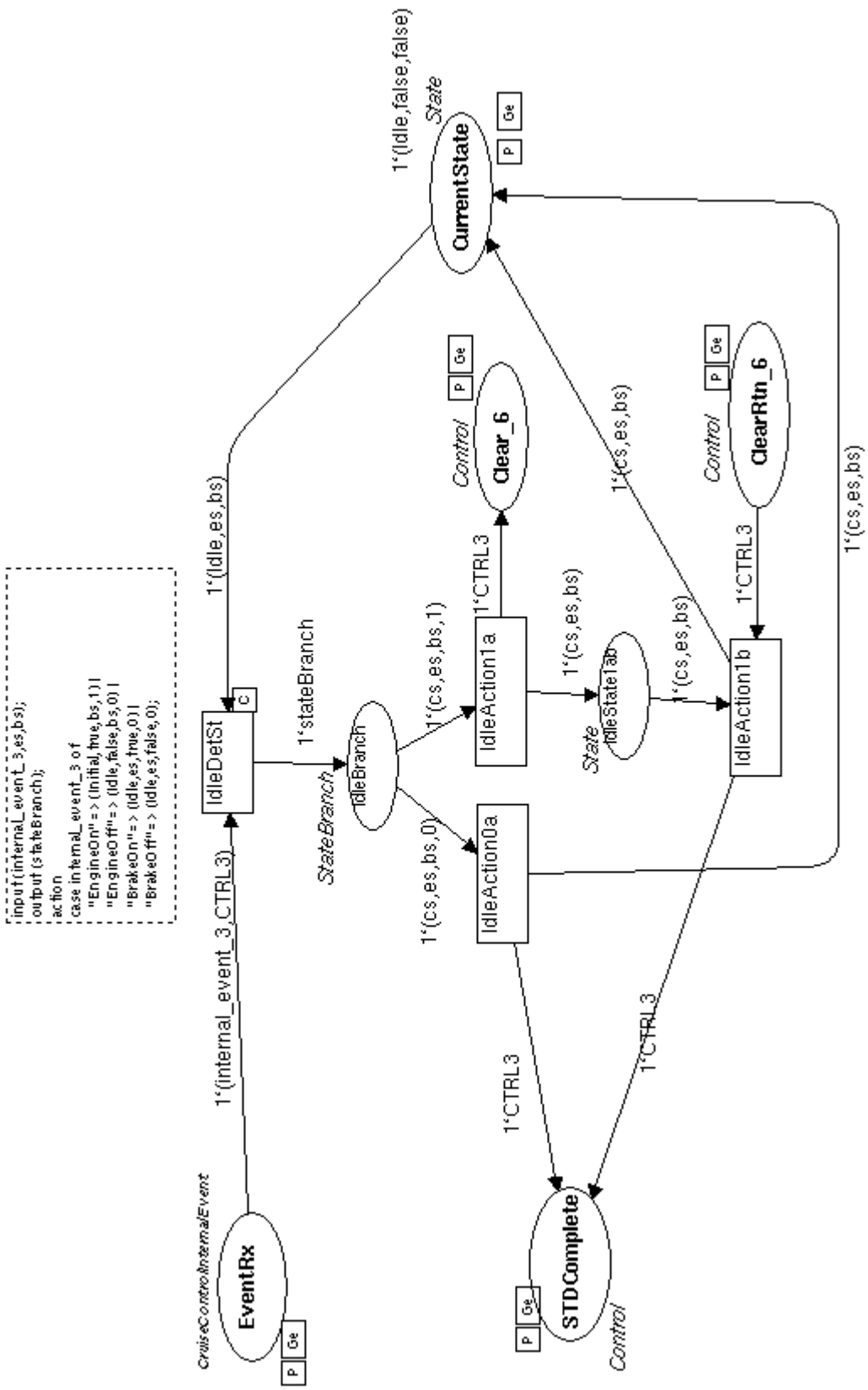


Figure 8. Cruise Control: Idle

5.1.2 Accelerating State

The behavior captured by the Accelerating state is slightly more significant than that provided by the Idle state. While in the Accelerating state, the automobile is accelerating to reach the desired cruising speed. When a "Cruise" request is detected, the system transitions from the Accelerating state to the Cruising state. During this transition, the Desired Speed is selected and (as indicated by the "Do" activity of the Cruising state) a "Maintain Speed" command is sent to the *SpeedControl* object. The other possible transitions from the Accelerating state are for the driver to press the brake, thus disengaging the cruise control and transitioning to the Cruising Off state or the engine stops and the system transitions to Idle.

The CPN model for the Accelerating state is shown in Figure 9. As with the Idle state, we again enter the model at a CPN transition (*AccelDetSt*) that uses a code segment to determine the new state and the action branch used to produce any desired actions. The only transition that produces actions from the Accelerating state is a transition to the Cruising state. Thus, this transition is assigned action branch 1 while all other inputs are assigned branch 0 and simply update the state and current engine and/or brake conditions.

Proceeding through branch 1, the first action is to select the desired speed. Thus, the *AccelAction1a* transition sends a control token to the CPN place corresponding to the Select Desired Speed operation call. The internal state information is transitioned to the *AccelState1ab* place and the *AccelAction1b* transition then waits for the Select Desired Speed operation to return as indicated by a control token in the *Select_6* place. Once this has completed, the state information is passed to the *AccelState1bc* place and the *AccelAction1c* transition performs the next action by sending a "MaintainSpeed" synchronous message to the *SpeedControl* Message buffer. Since this is a synchronous message, the *AccelAction1c* transition must first retrieve a *Free* token from the buffer, thus indicating that the transition is free to place a message in the *Message* place of the buffer. The *AccelAction1c* transition then sends the *ccCommand* token (set to "MaintainSpeed") to the *Message* place and passes the internal state information to the *AccelState1cd* place. The *AccelAction1d* transition must now wait for *SpeedControl* to retrieve the message from the buffer (since this is modeling synchronous communication). Once *SpeedControl* retrieves the message, it places a control token in the *Return* place and then *AccelAction1d* can fire, sending the updated state information to the *Current State* place and the control token to the *STDComplete* place. At this point, the processing for the Accelerating state is now complete and the *:CruiseControl* object may process the next event.

5.1.3 Modeling the Remaining Cruise Control States

The remaining cruise control states can be modeled by decomposing the remaining CPN transitions from Figure 7 using the same systematic approach used to model the Idle and Accelerating states presented above. Once the remaining states have been captured at this level, the *:CruiseControl* object may be analyzed using a tool such as DesignCPN [7]. As mentioned earlier, this analysis may occur independently by using CPN places to simulate the inputs and outputs. Alternately, the models may be integrated with the larger cruise control architecture (by using the actual CPN places corresponding to message queues, operation calls, etc.) to complete the behavioral modeling and analysis of the system as a whole. This analysis may cover such aspects as deadlock detection, performance analysis against time constraints, or checking boundary conditions to, for example, determine if the set queue sizes can handle the observed volume of message traffic.

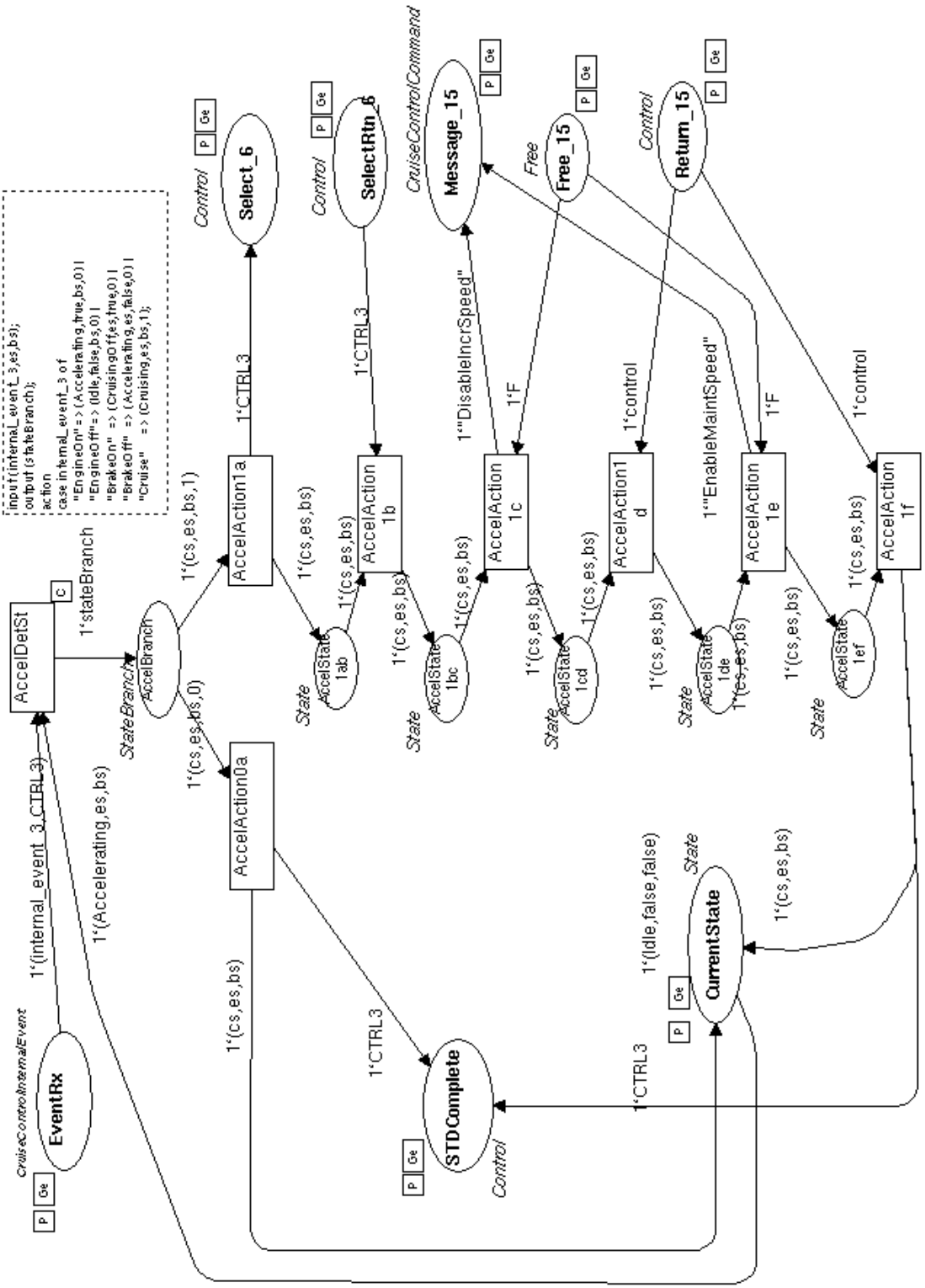


Figure 9. Cruise Control: Accelerating

6 Conclusions and Future Research

This paper presents a systematic, scaleable, and repeatable approach for using colored Petri nets to model the behavior of UML objects containing statecharts, which is capable of being automated. This approach may be integrated into the larger effort of modeling the overall concurrent software architecture using CPNs. The resulting CPN is then used to validate such dynamic properties as the absence of deadlock and starvation conditions as well as providing a timing and behavioral analysis of the architecture through simulation. This analysis through CPNs reduces the overall risk of software implementation by allowing behavioral characteristics to be validated from an architectural design rather than waiting for the system to be coded.

This paper represents on-going research efforts to integrate colored Petri nets with object-oriented software design methods for concurrent and real-time systems. Future research will explore the automatic generation of CPNs from UML. It is the goal of this continuing research to arrive at a set of CPN translation rules that can be effectively integrated with software design methods to provide increased reliability and analytical capabilities at multiple levels of abstraction.

7 References

[1] Rumbaugh, J., Jacobson, I., and Booch, G., *The Unified Modeling Language Reference Manual* Reading, Mass.: Addison-Wesley, 1999.

[2] Booch, G., Rumbaugh, J., and Jacobson, I., *The Unified Modeling Language User Guide* Reading, Mass.: Addison-Wesley, 1999.

[3] Pettit, R. G. and Gomaa, H. “Validation of Dynamic Behavior in UML Using Colored Petri Nets.” *UML 2000 Behavioral Semantics Workshop.* York, England. October, 2000.

[4] Gomaa, H., *Designing Concurrent, Distributed, and Real-Time Applications with UML*, Addison Wesley Object Technology Series, 2000, http://www.aw.com/cseng/titles/0-201-65793-7.

[5] David, R. and Alla, H., "Petri Nets for Modeling of Dynamic Systems: A Survey," *Automatica*, vol. 30, no. 2, pp. 175-202, 1994.

[6] Jensen, K., *Coloured Petri Nets: Basic Concepts, Analysis Methods, and Practical Use* Berlin, Germany: Springer-Verlag, 1997.

[7] DesignCPN. http://www.daimi.aau.dk/designCPN/.

[8] Harel, D., "On Visual Formalisms." CACM 31, 5 (May 1988), 514-530.

[9] Harel, D. and E. Gary, "Executable Object Modeling with Statecharts", Proc. 18th International Conference on Software Engineering, Berlin, March 1996

[10] Harel, D. and M. Politi, "Modeling Reactive Systems with Statecharts", New York, NY: McGraw-Hill, 1998.

A Calculus of Petri Net Components

N. Sidorova, M. Voorhoeve [1], J.C.S.P. van der Woude
Eindhoven University of Technology

Abstract

A framework is developed for modeling concurrent systems. The modeled systems are "open" in the sense that they can interact in prescribed ways with an unspecified environment. Systems are represented both as labeled Petri nets (allowing validation) and algebraic terms (allowing analysis).

1 Introduction

The engineering of systems is often performed by combining *requirements* and *models*. Requirements can be formulated in both natural and formal languages. It is almost impossible to describe a system by requirements alone. Their soundness can be validated by end users, but not their completeness: most likely systems can be constructed satisfying the requirements that are nonetheless unacceptable to the end user. This is the reason why requirements are complemented by models.

The systems we consider in this paper consist of many cooperating, yet independent asynchronous components. Petri Nets are well suited for modeling such systems. Advantages are its graphical nature and the decomposition of both states and events into local places and transitions. These properties make Petri nets an excellent tool for the validation of models by non-technical end users. Many state-of-the-art modeling techniques use Petri nets (or some derivative) for describing concurrent activities.

It stands to reason that Petri nets have been combined with hierarchy by isolating subnets and combining them to form large supernets. Examples are CPN [9] and ExSpect [8]. The common way of defining the semantics for these hierarchical approaches is to "flatten" the hierarchy and present the semantics of the complete system. So subnets obtain a semantics only in the context they were used in, which means that - in spite of efforts to make them reusable - they cannot be regarded as true components. True Petri net components must have a compositional semantics of their own. Compositionality means that components with the same semantics (i.e. mapped to the same mathematical object) must behave the same in any context.

In this paper we develop a framework for Petri net components, inspired by the theory behind process algebra [1]. Components are regarded as algebraic terms specifying subcomponents, their connection and the external interface. These terms are "open"; the interface fixes their possible uses by any environment. This results in a calculus with operators for connecting components, like fusion, relabeling and hiding of places and transitions, very similar to [12]. Each algebraic term can be normalized, resulting in a term containing the merge of basic components (places and transitions) that are connected by arc addition and then relabeled. These normalized

[1] email: wsinmarc@win.tue.nl

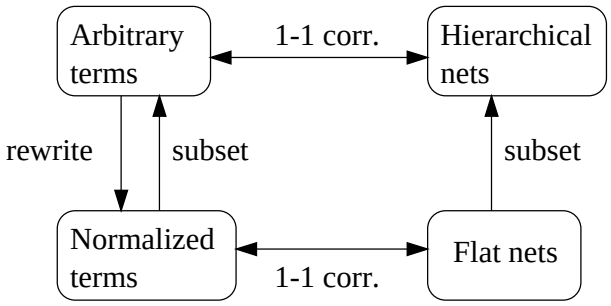


Figure 1: Correspondences between nets and terms

terms can be represented as "flat" Petri nets with partially labeled places and transitions (compare Figure 1). The state/event duality typical of Petri nets has been preserved: one can choose to label only places (reasoning only about states and their successions) or transitions (reasoning about possible and actual events) or combine the two.

The semantics of our component terms is indeed compositional: the (strong) bisimilarity equivalence is a congruence w.r.t. our operators. We can use it to speed up verifications of closed hierarchical nets by exploiting the hierarchy. However, the most important contribution is the possibility to define "open" subnets (components) that can be used and reused in arbitrary contexts.

This paper is a development from [12] (the calculus) and [2] (the semantics: branching instead of linear time). Both predecessor papers address P/T nets only, whereas the present paper applies to colored nets as well, thus providing a foundation for full-fledged component specification languages.

We have structured the paper as follows. In section 2, after some preliminary notations, we give some general definitions and results about transition systems with events having a baglike structure (i.e. they can occur simultaneously). In section 3, the events are pinned down further to consumption-firing-production triples typical of Petri nets. We define atoms (unconnected marked places and transitions) and operators for connecting them. In section 4 this calculus is specialized still further to cover "standard" colored nets. An example script is given in section 5 to illustrate its use as a specification and design language, after which some concluding remarks are given in section 6.

Acknowledgements: We thank our colleagues Tim Willemse, Twan Basten and Sjouke Mauw for their support and suggestions, as well as the anonymous referees.

2 Basic Notions

Let A, B be sets. A binary relation between A and B is a subset of $A \times B$. For relations R, S, R^{-1} denotes the converse of R and $R \,.\, S$ denotes the composition of R and S. Subclasses of binary relations are the sets of total ($A \to B$) and partial ($A \nrightarrow B$) functions. Function application of f to a is denoted as $f.a$. We omit the dot operator if this does not cause confusion, writing e.g.

$f(a, b)$ instead of $f.(a, b)$. If r is a relation and X a set, then $r[X] = \{a \mid \exists x \in X : x\ r\ a\}$.

The set of bags with base A is denoted as $\mathbb{B}.A$. For $\alpha \in \mathbb{B}.A$, $a \in A$, $w_\alpha a$ denotes the weight of a in α. The carrier $C.\alpha$ of $\alpha \in \mathbb{B}.A$ is the set $\{a \mid w_\alpha a > 0\}$. We identify bags $\alpha \in \mathbb{B}.A$ and $\beta \in \mathbb{B}.B$ when $C.\alpha = C.\beta$ and $\forall c \in C.\alpha : w_\alpha c = w_\beta c$. If $\alpha \in \mathbb{B}.A$, $\beta \in \mathbb{B}.B$, then $\alpha + \beta$ is the element of $\mathbb{B}(A \cup B)$ satisfying $w_{\alpha+\beta}x = w_\alpha x + w_\beta x$. The empty bag is denoted by 0 (its carrier is the empty set). If no confusion can arise, we can interpret a set A as a bag with carrier A and weights 1 throughout. Note that $(\mathbb{B}.A, +, 0)$ is a commutative monoid.

If $r \in A \nrightarrow B$, then $\bar{r} \in \mathbb{B}.A \to \mathbb{B}.B$ satisfies $w_{\bar{r}.\alpha}b = \Sigma(w_\alpha a \mid r.a = b)$. So if r is the empty function, the range of $\bar{r}$ is $\{0\}$.

We shall define transition systems and the bisimilarity equivalence relation on them. Bisimilarity abstracts as much as possible from states; two states are equivalent iff they allow the same actions leading to equivalent states. We presuppose a universe of *event labels* $\mathcal{E}$.

Definition 1 *A* transition system *is a pair (S, E), where S is a state space and $E \in \mathcal{E} \to \mathbb{P}(S \times S)$ a function assigning state relations to events. The relations $E.e$ are denoted by $\xrightarrow{e}$. If $C \xrightarrow{e} C'$, there exists an event with label e leading from C to C'.*

Given a transition system (S, E), a relation $Q \subseteq (S \times S)$ is a simulation *iff for every $e \in \mathcal{E}$ we have $Q^{-1} \,.\, \xrightarrow{e} \subseteq \xrightarrow{e} \,.\, Q^{-1}$. A simulation Q is a* bisimulation *iff Q^{-1} is a simulation too. States C, D are called* bisimilar *iff there exists a bisimulation Q such that $C\ Q\ D$.*

A component *is an equivalence class of bisimilar states in a transition system.*

Note that we use a single transition system S in the definition, which is enough for our purposes. A definition with several transition systems is not hard, but somewhat more cumbersome. A simulation is a relation Q that satisfies the transfer property: if $C\ Q\ D$ and $C \xrightarrow{e} C'$, then there exists a D' such that $D \xrightarrow{e} D'$ and $C'\ Q\ D'$. A weaker equivalence relation is *branching bisimilarity* [6]. This equivalence relation is most appropriate for comparing implementations to their specification. The bisimilarity notion above is often called "strong" bisimilarity in contrast.

The state spaces of our transition systems will consist of algebraic terms describing the events that can occur to its states. Those terms are given via SOS production rules (structured operational semantics [11]). These rules define the events that can occur in a given state; it is implicitly assumed that no event can occur in a given state unless there is a production rule defining it (no junk property). The syntax is $\dfrac{p, q, \ldots}{r}$, where $p, q, \ldots$ are the premisses and r the conclusion.

We suppose that the set of event labels $\mathcal{E}$ is a commutative monoid with addition operator $+$ and neutral element 0. The interpretation of $e + f$ is the simultaneous occurrence of events e and f. The neutral element 0 indicates the absence of any visible event. Every component C will satisfy $C \xrightarrow{0} C$. Note that it is possible that $C \xrightarrow{0} C'$ with $C \neq C'$: the state may change without any event being visible.

The binary free merge operator $C \,||\, D$ allows the combination of components; there exist various transformation operators Θ_R, with R a binary or ternary relation between events. In the sections to come, we will specialize these transformation operators in various ways.

We assume the existence of a set A of atoms with a set of relations $\xrightarrow{e}_A$ between them. Our state space S is the smallest set satisfying

1. $A \subseteq S$
2. $C, D \in S \Rightarrow C \,||\, D \in S$
3. $(R \in \mathcal{E}^2 \cup \mathcal{E}^3 \wedge C \in S) \Rightarrow \Theta_R.C \in S$.

The relations $\xrightarrow{e}$ are the smallest subsets of S^2 containing the $\xrightarrow{e}_A$ or produced from the SOS rules given below.

$$\frac{C \xrightarrow{e} C',\ D \xrightarrow{f} D'}{C \,||\, D \xrightarrow{e+f} C' \,||\, D'} \qquad \frac{C \xrightarrow{e} C',\ e\ R\ f}{\Theta_R.C \xrightarrow{f} \Theta_R.C'} \qquad \frac{C \xrightarrow{d} \cdot \xrightarrow{e} C',\ (d, e, f) \in R}{\Theta_R.C \xrightarrow{f} \Theta_R.C'}$$

From the structure of these rules, it can be deduced that the merge and transformation operators are congruences w.r.t. both strong and branching bisimilarity (c.f. [5]).

Theorem 1 *Let B, C, D be components and $R, T \subseteq \mathcal{E}^2$. Then*
$C \,||\, D = D \,||\, C, \quad (B \,||\, C) \,||\, D = B \,||\, (C \,||\, D), \quad \Theta_R \cdot \Theta_T = \Theta_{T.R}$

Proof: We shall derive the first of these equations in order to illustrate the type of reasoning used here. We construct the relation $Q = \{(C \,||\, D, D \,||\, C) \mid C, D \in S\}$ between terms. It is sufficient to prove that Q is a bisimulation. We must prove two transfer properties; the first is that $X\ Q\ Y$ and $X \xrightarrow{e} X'$, there exists an Y' such that $Y \xrightarrow{e} Y'$ and $X\ Q\ Y'$. Since $X\ Q\ Y$, X must be of the form $C \,||\, D$ and Y of the form $D \,||\, C$. We have $C \,||\, D \xrightarrow{e} X'$ and as the only SOS rule matching the lhs is the first, X' must have the form $C' \,||\, D'$ for some C', D' satisfying $C \xrightarrow{f} C'$, $D \xrightarrow{g} D'$ and $e = f + g$. It is now easy to find a Y', namely $D' \,||\, C'$ satisfying the transfer property. Since Q is symmetric, the other transfer property is identical to the first, so Q is indeed a bisimulation.

The other two equations are derived similarly. □

3 Global Petri Nets

As a first step towards formalizing Petri net components, we decompose events into consumption, firing and production. Unlike colored or P/T nets, firings are not associated to distinct transitions nor consumptions/productions to distinct places, whence the term “global”.

In Figure 2, the main idea is illustrated for defining the semantics of net components, namely the use of consumption/firing/production triples. Three components C, D, E are shown in the figure. The component C can be transformed into D by external production of a token in place p. By externally consuming the same token, D becomes C. By a firing step, D becomes E.

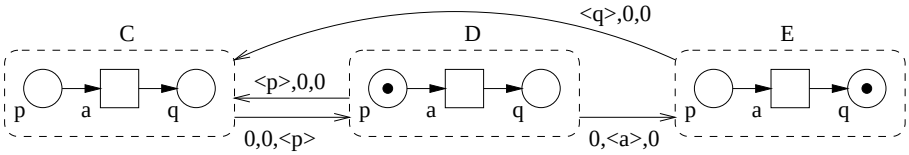


Figure 2: Illustration of events

Internal consumptions and productions (i.e. associated to a firing step) go with the firing and are not separately indicated.

This way of associating a transition system to nets allows both state-based and action-based reasoning. We can discover that there is one and only one token in p-labeled places of D, since one p-token (and not more) can be externally consumed from it. We can also discover that C contains an a-labeled transition that is not enabled but becomes so by externally adding a p-token, and which internally consumes a p token and produces a q token, since the p-token that was there in D has gone and a q-token has appeared after firing.

So we have indeed disguised state-based properties (the presence and absence of tokens) as events (possible and impossible consumptions). Of course, we can have several external consumptions, firings and external productions concurrently, if there are enough tokens to allow the consumptions and firings. Tokens that are produced (either directly or by the firings) have to wait for the next step to be used.

We will indicate how this idea has been transformed into a calculus. Our atoms are unconnected marked places, that allow production and consumption only, and unconnected transitions that allow firing only. The atoms can be *merged* and by *arc addition* they can become connected, associating internal productions and/or consumptions to firings. By *relabeling*, we can then redefine the component's interface. This relabeling allows e.g. to hide tokens (disabling external consumptions and productions), hide firings (relabeling them to 0) and introduce nondeterminism (giving different tokens/firings the same label).

The above atoms and operators suffice to model terms that have a direct representation as partially labeled Petri nets. Their application to arbitrary components causes a notion of hierarchy. Finally, the operators are supplemented with a *place fusion* operator, extending the options for connecting components. We can rewrite any term to a normal form that has a flat net representation.

We shall now formalize the notions above. We first set $\mathcal{E} = \mathcal{O} \times \mathcal{A} \times \mathcal{O}$, where $\mathcal{O}, \mathcal{A}$ are commutative monoids (place label, resp. action bags). We assume the set of labels to be infinite, so "fresh" labels are always available. We set $(a, b, c) + (d, e, f) = (a + d, b + e, c + f)$ so $\mathcal{E}$ is a commutative monoid.

We call a relation r between commutative monoids *right linear* iff (1) $0\ r\ 0$, (2) $(a\ r\ b \wedge c\ r\ d) \Rightarrow (a+c)\ r\ (b+d)$ and (3) $a\ r\ (b+c) \Rightarrow \exists a_1, a_2 : a = a_1 + a_2 \wedge a_1\ r\ b \wedge a_2\ r\ c$. It is *right normal* if $0\ r\ x \Rightarrow x = 0$ and *left normal* if r^{-1} is right normal.

From now on, let $A \subseteq \mathcal{A}$, $I \in \mathcal{O}$, $r \in \{s \cup t \mid s \subseteq \mathcal{O}^2 \wedge t \subseteq \mathcal{A}^2\}$ right linear and right normal, $v, w \subseteq (\mathcal{O} \times \mathcal{A})$ both right linear and left normal, $U \subseteq \mathcal{O}$ with $0 \in U$.

Definition 2 *The components P_I (place), T_A (transition) and the operators λ_r (renaming), $\gamma_{v,w}$ (arc addition) and ϕ_U (place fusion) are defined by the SOS rules below.*

$$1 \quad \frac{x \in A}{T_A \xrightarrow{0,x,0} T_A} \qquad 2 \quad \frac{c \leq I}{P_I \xrightarrow{c,0,p} P_{I-c+p}} \qquad 3 \quad \frac{C \xrightarrow{c,s,p} C',\ c\ r\ d,\ p\ r\ q,\ s\ r\ t}{\lambda_r.C \xrightarrow{d,t,q} \lambda_r.C'}$$

$$4 \quad \frac{C \xrightarrow{d+c,s,p+q} C',\ d\ v\ s,\ q\ w\ s}{\gamma_{v,w}.C \xrightarrow{c,s,p} \gamma_{v,w}.C'} \qquad 5 \quad \frac{C\ (\xrightarrow{x,0,x} \cdot \xrightarrow{c,s,p})\ C',\ x \in U}{\phi_U.C \xrightarrow{c,s,p} \phi_U.C'}$$

, Note the non-atom operators being special cases of Θ_R, e.g. $\gamma_{v,w} = \Theta_R$ with $(c,\ s,\ p)\ R\ (c',\ s',\ p')$ iff $s = s' \wedge (c - c')\ v\ s \wedge (p - p')\ w\ s$. The operator ϕ_U is the one with a ternary R.

We will discuss the notions behind this formal definition.

1. Unconnected transitions (T_A) can execute any step from its parameter set A without any state change.

2. Unconnected marked places (P_I) allow an arbitrary consumption of tokens c contained in the marking I ($c \leq I$) and an arbitrary production p, resulting in a new state $I - c + p$.

3. The renaming operator λ_r allows to rename steps, consumptions and productions by relation r. In most cases, the relation r will be a partial function. Hiding of steps is done by renaming them to 0. The right linearity of q ensures that hidden steps cannot become visible again. To allow the hiding of tokens, we can use a partial function r to shield tokens outside dom.r from external production or consumption.

4. The arc addition operator, when used with functions instead of relations, states that an event $(0, s, 0)$ (i.e. a step) between $\gamma_{v,w}.C$ and $\gamma_{v,w}.C'$ will be possible iff the event $(v.s,\ s,\ w.s)$ existed between C, C' before the arc addition. A special choice of the v, w relations yields the blocking operator ξ_E for certain $E \subseteq \mathcal{A}$ with the production rule $\dfrac{C \xrightarrow{c,s,p} C',\ s \in E}{\xi_E.C \xrightarrow{c,s,p} \xi_E.C'}$.

5. The fusion rule states that certain states that *look* the same in fact *become* the same in ϕ_U. So if an event $(c,\ s,\ p)$ is possible from a state C'', the same event is possible from C, provided C'' can be reached from C by a token switch: consuming and producing tokens that both are labeled x (but may differ).

Component terms are recursively defined from the places, transitions and merge and transformation operators.

Definition 3 *The set $\mathcal{C}$ of* component terms *is the smallest set satisfying:*

1. $T_A,\ P_I \in \mathcal{C}$,

2. *if* $C,\ D \in \mathcal{C}$, *then* $C \mid\mid D \in \mathcal{C}$,

3. *if* $C \in \mathcal{C}$, *then* $\lambda_r.C, \xi_E.C, \gamma_{v,w}.C, \phi_U.C \in \mathcal{C}$.

We shall now derive rules by which the terms in $\mathcal{C}$ can be rewritten to a flat normal form. The place fusion and blocking operators can be eliminated altogether and the other operators can be given a certain order. We start with some basic equations, stating that the application order of most operators can be reversed.

We define the sum of sets and relations $A, B \subseteq \mathcal{A}$ by $A + B = \{a + b \mid a \in A, b \in B\}$. If r, s are both right linear, then so is $r + s$.

Theorem 2 *The following equations hold.*

$$\lambda_r.T_A = T_{r[A]} \quad \gamma_{v,w}.T_A = T_{A\cap(v^{-1}\cap w^{-1})[0]} \quad \phi_U.T_A = T_A \quad T_A \,||\, T_{A'} = T_{A+A'}$$
$$\lambda_r.P_I = P_{r[I]} \quad \gamma_{v,w}.P_I = P_I \quad \phi_U.P_I = P_I \quad P_I \,||\, P_J = P_{I+J}$$
$$\lambda_r \cdot \lambda_x = \lambda_{r.x} \quad \gamma_{v,w} \cdot \lambda_r = \lambda_r \cdot \gamma_{r.v.r^{-1},r.w.r^{-1}} \quad \gamma_{v,w} \cdot \gamma_{x,y} = \gamma_{v+x,w+y}$$
$$\phi_U \cdot \lambda_r = \lambda_r \cdot \phi_{r^{-1}[U]} \quad \phi_U \cdot \gamma_{v,w} = \gamma_{v,w} \cdot \phi_U$$

Proof: We construct the obvious bisimulation between terms in each case. □

We can eliminate bijective renaming.

Theorem 3 *If* r *is a bijection, then* λ_r *is a bijection on* $\mathcal{C}$ *with inverse* $\lambda_{r^{-1}}$. *We also have the equation* $\lambda_r \cdot \gamma_{v,w} = \gamma_{r.v,r.w}$ *in this case. For all* C, D, *we have* $\lambda_r(C \,||\, D) = \lambda_r.C \,||\, \lambda_r.D$.

Proof: By Theorem 2, $\lambda_r \cdot \lambda_{r^{-1}} = \lambda_{r.r^{-1}}$, which is the identity. Again, bisimulations are constructed for each equation. □

We can now prove the flat net theorem: each hierarchical net ($\mathcal{C}$ term) can be represented as the renaming of connected places and transitions.

Theorem 4 *Each term* $C \in \mathcal{C}$ *can be rewritten to the normal form* $(\lambda_r \cdot \gamma_{v,w}).(T_A \,||\, P_I)$.

Proof: We use structure induction to terms. Use the equations in Theorem 2 as rewrite rules from left to right, proving the theorem for all terms of the form $(f \cdot g).C$, with f, g transformations. We are left with terms $C \,||\, D$. By the induction hypothesis, we may assume that C, D are in normal form, so let $C = (\lambda_u \cdot \gamma_{v,w}).(T_A \,||\, P_I)$. With $f = \lambda_u$, $E = \gamma_{v,w}.(T_A \,||\, P_I)$, we can find a fresh set of labels and a bijective renaming r such that $\lambda_r \cdot f = f \cdot \lambda_{r^{-1}} = f$. We then have $C \,||\, D = f.E \,||\, D = f.E \,||\, \lambda_r \cdot \lambda_{r^{-1}}.D = (\lambda_r \cdot f).(E \,||\, \lambda_{r^{-1}}.D)$. Since r was injective, we can eliminate the $\lambda_{r^{-1}}$ by Theorem 3, obtaining the term $(\lambda_r \cdot f).(E \,||\, D')$, with D' in normal form. We repeat the same trick with $f = \gamma_{v,w}$, $E = T_A \,||\, P_I$ to obtain $(\lambda_{r'} \cdot \gamma_{v,w})E \,||\, D''$. Repeating this for D'', we obtain a merge of places and transitions, that can be rewritten to the merge of one place and one transition by Theorem 2. □

4 Local Petri nets

In this section we further specialize the operators of the previous section to arrive at more or less "standard" colored nets. The firings that can occur are localized into distinct transitions and the tokens that are consumed and produced are localized into distinct places.

We attach a label to every place; tokens possess the label of their place and a value in addition. A marked place is parametrized by a bag of such tokens (with the same label). We attach a function of labels to values (a "record") to every transition firing; a transition is parametrized with a set of such records (with the same domain) (a "table").

The selection of operators is a compromise between power and clarity, with CPN in mind. It is not clear whether a closer approximation of CPN is possible or worthwhile. Arc addition is split into production and consumption. We choose one label-value pair of a firing record to go with a label-value pair of a token.

We define fusion based on labels only: all tokens from places labeled with a fusion label are dumped in the same place (as it were). Relabeling is split into transition and place relabeling and does not change values. Blocking is split into encapsulation: the blocking of any step containing a forbidden label-value pair and synchronization: the blocking of any step that does not combine the label-value pairs to be synchronized. Hiding is split into place hiding and transition hiding and is based on labels.

The formal definition of tha atoms and operators of our global net calculus involves rather awkward notation. To get an impression of them, the reader is invited to compare the example script in the next section. We set the monoids $\mathcal{O}$ and $\mathcal{A}$ to be bags of label-value pairs: $\mathbb{B}.(\mathcal{L} \times \mathcal{V})$, where $\mathcal{L}$ is an infinite set of labels and $\mathcal{V}$ a set of values. The *closure* of a set F of functions is set of bags that can be written alpha as a finite sum $\sigma\, f_i$ with the $f_i \in F$ are interpreted as bags with weights 1.

Definition 4 *We define*

1. *For $A \subseteq (L \to \mathcal{V})$, with $L \subseteq \mathcal{L}$ finite, we set $T[A] = T_B$, with B the closure of A.*
2. *For $\ell \in \mathcal{L}, \alpha \in \mathbb{B}.\mathcal{V}$, we set $P[\ell, \alpha] = P_I$, where I is the bag with carrier $\ell \times C.\alpha$ such that $w_I(\ell, v) = w_\alpha v$ for $v \in \mathcal{V}$.*
3. *For $f \in (\mathcal{L} \times \mathcal{L}) \nrightarrow \mathbb{P}(\mathcal{V} \times \mathcal{V})$, we set $\chi[f] = \gamma_{v,0}$ and $\pi[f] = \gamma_{0,v}$, with $v = \{((\ell, x), (m, y)) \mid x\ f(\ell, m)\ y\}$ (consumption/production). The function 0 has the empty bag as range.*
4. *For a relation $g \subseteq \mathcal{L} \times \mathcal{V}$, we define $\partial[g] = \xi_B$, with $B = \mathbb{B}.(\mathcal{L} \times \mathcal{V} \setminus g)$ (encapsulation) and $\sigma[g] = \xi_B$, with B the closure of $\{h \subseteq g \mid \mathrm{dom}.h = \mathrm{dom}.g \wedge h \text{ is functional}\}$ (synchronization).*
5. *For $L \subseteq \mathcal{L}$ finite, we define $\phi[L] = \phi_U$, with $U = L \times \mathcal{V}$.*
6. *For $h \in \mathcal{L} \nrightarrow \mathcal{L}$, let $H = \mathrm{dom}.h$. We define $\theta[h] = \lambda_{1_{\mathcal{L}\times\mathcal{V}}, \bar{q}}$, with 1_A the identity function on A and $q = (h \cup 1_{\mathcal{L}\setminus H}) \times 1_{\mathcal{V}}$ (transition relabeling, which is total!) and $\eta[h] = \lambda_{\bar{r},1}$, with $r = h \times 1_{\mathcal{V}}$ (place relabeling, which is partial).*
7. *For $H \subseteq \mathcal{L}$, we define $\tau[H] = \lambda_{1,\bar{q}}$, with $q = id_{\neg H} \times \mathcal{V}$ (transition hiding) and $\upsilon[H] = \lambda_{\bar{r},1}$, with $r = id_{\neg H} \times \mathcal{V}$ (place hiding).*

We define labeled components, terms defining hierarchical colored Petri nets.

Definition 5 *A labeled component is either a term $T[A]$ or $P[\ell, \alpha]$ or $C \,||\, D$ with C, D labeled components or a $F.C$ with $F \in \{\chi[f], \pi[f], \partial[g], \sigma[g], \phi[L], \eta[h], \theta[h], \upsilon[H], \tau[H]\}$ and C a labeled component.*

Like in the previous section, labeled components can be normalized. Because of the added complexity, the normal form is less clear-cut. Essentially it consists of the same order: hiding, then renaming, then a bunch of consumptions/productions applied to the merge of several places and transitions. Minor extensions and alterations of our operators may lead to a more or less smooth normal form. The ideas behind the normalization have been introduced in the previous section: we establish the way that the transformations "commute" if they are in the "wrong" order and use the "fresh label renaming trick" to invert the order of transformation and merge.

A language is still needed for representing the functions, sets and relations between values. There exist many languages that will fit the bill nicely.

5 Example

We can represent components textually by scripts that define functions, sets and relations to go with the operators above. A tentative example script is given below.

```
E[action n, action p] :=
 hide[place x, action c] in encap[c with c <> 0] in
  sync[c with c=0 with n] in fuse[x] in
   (C[place y renamed to x, action c] merged to
    D[place x, action n,p]);
C[place y:int, action c:nat] :=
 hide[action d,e] in consume[d from y with d=y] in
   produce[c to y with c=y, e to y with e=y] in
    (trans[d:int, e:int with e-d < 2 and d-e < 2] merged to
     trans[c:nat] merged to place[y:int init <0>])
D[place x:int, action n,p] :=
 consume[p from x with x>0, n from x with x<0] in
   (trans[p] merged to trans[n] merged to place[x:int init empty)])
```

The semantics of this script is as follows:
$E = (\upsilon[\{\mathsf{x}\}] \,.\, \tau[\{\mathsf{c}\}] \,.\, \partial[\{(\mathsf{c}, \mathbb{N} \setminus \{0\})\}] \,.\, \sigma[\{(\mathsf{c}, \{0\}), (\mathsf{n}, \mathcal{V})\}] \,.\, \phi[\{\mathsf{x}\}]).(\eta[\{(\mathsf{y}, \mathsf{x})\}].C \,||\, D)$, with
$C = (\tau[\{\mathsf{d}, \mathsf{e}\}] \,.\, \chi[\{((\mathsf{d}, \mathsf{y}), \{(d, y) \mid d = y\})\}] \,.\, \pi[f]).M$ and
$M = (T[\{\{(\mathsf{d}, d), (\mathsf{e}, e)\} \mid d, e \in \mathbb{Z} \wedge -2 < d - e < 2\}] \,||\, T[\{\{(\mathsf{c}, c)\} \mid c \in \mathbb{N}\}] \,||\, P[\mathsf{y}, \langle 0 \rangle]$ and
$f = \{((\mathsf{c}, \mathsf{y}), \{(c, y) \mid c = y\}), ((\mathsf{e}, \mathsf{y}), \{(e, y) \mid e = y\})\}$ and
$D = (\chi[\{((\mathsf{p}, \mathsf{x}), \{(p, x) \mid x > 0\})\}] \,.\, \chi[\{((\mathsf{n}, \mathsf{x}), \{(n, x) \mid x < 0\})\}]).N$ and
$N = T[\{\{(\mathsf{p}, \bullet)\}\}] \,||\, T[\{\{(\mathsf{n}, \bullet)\}\}] \,||\, P[\mathsf{x}, 0]$.

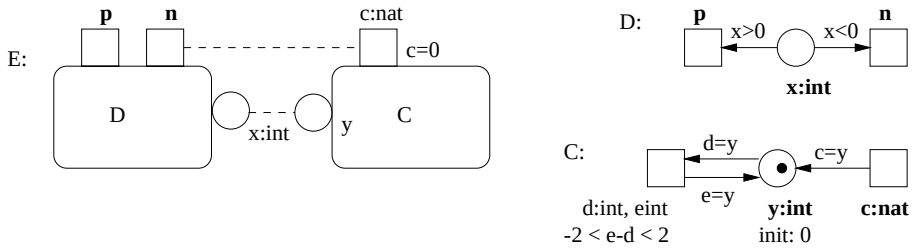


Figure 3: Colored Petri Net of example script

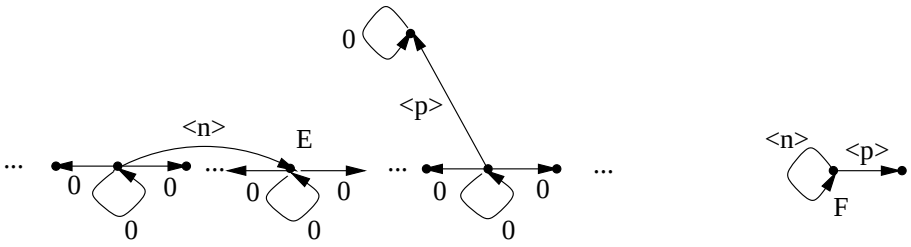


Figure 4: Transition system and reduction of example

Here the • signifies some value that has no importance. The identifiers in `typescript` font are to be interpreted literally (i.e. as labels), the others are placeholders for the definition of sets and functions. We can graphically represent this script by the CPN-like annotated net in Figure 3. There, the identifiers in boldface represent the external labels.

We can normalize the above script and arrive at the following script. We will forgo a semantics and graphical representation.

```
E[action n, action p] :=
 hide[place x, action d,e] in
 consume[d from x with d=x, p from x with x>0, n from x with x<0] in
   produce[n to x with x=0, e to x with e=x] in
    (trans[d:int, e:int with e-d < 2 and d-e < 2] merged to
     trans[n] merged to trans[p] merged to place[x:int init <0>])
```

The transition system of E is shown in Figure 4. Its reduction modulo branching bisimilarity is shown as F in the same figure.

6 Conclusion

This paper defines a semantics for colored hierarchical Petri nets, which supports both action-oriented and state-oriented approaches, matching the dual nature of Petri nets. The compositional nature of our framework allows the specification of components that can be implemented independently respecting the specified interface.

Our paper has been inspired by process algebra, more specifically by the wish to be able to reason about and manipulate with Petri nets like process algebra terms. When comparing our framework to a process-algebraic one like μCRL [7], one notices the same expressive power. However, the presence of true sequencing and choice operators in μCRL makes it better suited for specifying technical systems like data communication protocols, whereas the added connection possibilities, graphical nature and the step (instead of interleaving) semantics of our framework seem to give it a slight edge for programming-in-the-large.

We already discussed the relation of this paper to [2], [12]. It is also interesting to compare our approach to the Petri Box calculus originated by [3]. The Petri Box calculus is aimed at giving a Petri net (and thus causal) semantics for the language B(PN)^2 [4], [10], somewhat similar to μCRL. Our aim is to use colored net components as a high-level specification and design tool.

It may be worthwhile to extend our step semantics to more "truly concurrent" ones. This would allow e.g. an action refinement operator in our calculus, which is not compositional in step semantics. However, we hesitate to sacrifice the simplicity of transition systems. Our approach gives the modeler the option of preserving the causal connections between actions by not hiding all places (causes) between the transitions involved.

Much work is still needed to arrive at practical verifications, let alone "best practice" rules for deriving correct models from a set of requirements, but we believe that this paper is a serious step towards this goal.

References

[1] J.C.M. Baeten and C. Verhoef. Concrete Process Algebra. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 4, pages 149–268. Oxford University Press, Clarendon, UK, 1995.

[2] T. Basten and M. Voorhoeve. An Algebraic Semantics for Hierarchical P/T Nets. In G. De Michelis and M. Diaz, editors, *Proceedings ATPN '95*, volume 935 of *Lecture Notes in Computer Science*, pages 45–65, Torino, Italy, 1995. Springer–Verlag, Berlin.

[3] E. Best, R. Devillers, and J. Hall. The Petri Box Calculus: a New Causal Algebra with Multilabel Communication. In G. Rozenberg, editor, *Advances in Petri Nets 1992*, volume 609 of *Lecture Notes in Computer Science*, pages 21–69. Springer–Verlag, Berlin, 1992.

[4] E. Best and R.P. Hopkins. B(PN)^2 - a Basic Petri Net Programming Notation. In A. Bode, M. Reeve, and G Wolf, editors, *Proceedings PARLE '93*, pages 379–390. Springer–Verlag, Berlin, 1993.

[5] Wan Fokkink. Rooted branching bisimulation as a congruence. *Journal of Computer and System Sciences*, 60(1):13–37, 2000.

[6] R.J. van Glabbeek and W.P. Weijland. Branching Time and Abstraction in Bisimulation Semantics. *Journal of the ACM*, 43(3):555–600, 1996.

[7] J.F. Groote and M.A. Reniers. Algebraic Process Verification. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, pages 1151–1208. Elsevier Science BV, Amsterdam, 2001.

[8] K.M. van Hee. *Information Systems Engineering: a Formal Approach*. Cambridge University Press, Cambridge, 1994.

[9] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use.* EATCS monographs on Theoretical Computer Science. Springer–Verlag, Berlin, 1992.

[10] H. Klaudel. Compositional high-level Petri net semantics of a parallel programming language with procedures. *Science of Computer Programming* (to appear).
url: `www.univ-paris12/~lacl/klaudel/proc.ps.zip`.

[11] G.D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Dept., Aarhus University, 1981.

[12] L Priese and H Wimmel. A uniform approach to true-concurrency and interleaving semantics for Petri nets. *Theoretical Computer Science*, 206:219–256, 1998.

Steps Towards Formal Verification of Agent-based E-Business Applications

Nick Szirbik, Gerd Wagner
Dept. of Information & Technology, Eindhoven University of Technology
Postbus 513, 5600 MB Eindhoven
{g.wagner; n.b.szirbik}@tm.tue.nl

Abstract

In this preliminary report about ongoing work, we discuss the issues of formalizing the specification of interaction process types for e-business applications and identifying techniques for *proving correctness properties* for them. We introduce the concept of *coherence* for interaction processes in the broader context of multiagent communication based on agent communication languages. Interaction process types can be sepcified by means of reaction rules in a declarative way. In order to make use of the formal analysis techniques established for Petri nets, we investigate possibilities how to translate reaction rules into Petri nets.

1. Introduction

In a world characterized by rapid change, e-business applications have inherently a very short life-cycle, short time-to-market, must be adaptable for upgrade and have to follow the changes in the business processes that are supported by these application. Their development would be less error-prone if it would be possible to use formal verification techniques implemented with automated reaoning technologies. Formal verification is well suited for systems with a relatively small state space, or where the essential aspects of the system can be captured in a model that may be formally checked. To validate the final application, testing and other validation techniques are used. However, the validation phase can be extremely costly, and formal verification can help to avoid major errors in the design, reducing the costs of the validation phase. Also, verification can "catch" problems which are extremely difficult to be identified in the validation phase and usually appear late after the deployment of the application, incurring losses for the user and costly maintenance for the application provider.

We discuss in this paper the issue of *coherence* in interaction processes, which subsumes a certain set of properties that we claim to be important to be checked for e-business applications. The paper is structured around this concept; the next section explains relevant aspects of the agent paradigm, section 3 is presenting the coherence concept, section 4 is explaining the action/event and claim/commitment concepts and their usage in supporting coherence, section 5 is about the rule specification used in the paper, section 6 explains how some particular types of rules can be translated into Petri Nets, section 7 is presenting how inter-agent communication can be modelled, section 8 puts some of the questions that have been raised by this reasearch and section 9 concludes the paper. Some ideas above potential future work are also given there.

2. Agent-Oriented E-Business Modeling and the Issue of Verification

The objective of the research presented in this paper is to analyse the problem of, and develop a method for, how to verifiy the coherence of e-business processes as automated interaction processes across different orgaizations. For obtaining high-level concepts of interaction and communication, an agent-oriented approach to e-business modeling seems to be most promising. *Agent-Orientation* is emerging as a new paradigm in software and information systems engineering. It offers a range of high-level abstractions that facilitate the conceptual and technical integration of communication and interaction with established information system technology. Agent-Orientation is highly significant for business systems since business processes are driven by and directed towards agents (or actors - we treat both terms as synonyms), and hence have to comply with the physical and social dynamics of interacting individuals and organizations. While today's enterprise information system technology is largely based on the metaphors of data management and data flow, Agent-Orientation emphasizes the fundamental role of actors/agents and of communication and interaction for analysing and designing organizations and organizational information systems. This turns out to be crucial for a proper understanding of business processes and their underlying rules.

Interaction processes are determined by the behaviour of the participating agents. The behaviour of natural (and artificial) agents can be described (and specified) by means of rules. Business rules are the policies and constraints of the business, whether the "business" is banking, software development or health care. The Object Management Group simply defines them as "declarations of policy or conditions that must be satisfied". They can be expressed in ordinary language (possibly using special templates) or by means of a formal language. An interaction process type can be specified by defining, for each participating party, a set of inter-related rules. The formal language for such rules must include facilities for incorporating agent communication languages, message content languages, and domain ontologies in a flexible (suitably parameterized) manner.

There are two main questions to be answered:

1) what properties are worth and essential to be verified in such a system
2) from what details can we abstract away in order to make verification feasible (i.e. computationally tractable).

The verification method can employ different techniques.

E-business applications are supporting interaction processes between various types of enterprises (viewed as *institutional agents*). An interaction process can be defined in many ways, but we prefer here a definition based on the agent paradigm. We interpret this paradigm in a way that emphasizes that all the participating active entities are agents. The agents can be generally classified in five classes: human, institutional, software, robots and animals, but only the first three are interesting to our discussion here. E-business processes can be semi-automated or fully automated.

It is well-known that the process formalism of Petri Nets can be used for verifying certain properties of business processes [Bas98, Aal00, Aal01]. We discuss the possibility to translate a rule-based process specification into a Petri net An earlier attempt to translate rules into Petri Nets has been made in [Naz93].

3. Coherence in Interaction Processes

An interaction process is a temporally ordered set of *events*, *actions* and *activities* perceived and performed by agents, following a *protocol* that specifies the *type* of the interaction process. An interaction protocol can be specified by a set of *reaction rules* that determine the behavior of the involved agents. Business interactions, for example, comprise communications (message exchange), payment transactions, product deliveries, and the performance of services. An interaction process

may be fully automated, e.g., in electronic commerce supply chains, in agent-mediated group scheduling, in robotic soccer games (RoboCup), in Automatically Guided Vehicle (AGV) Systems, and in other types of artificial multiagent systems. In many cases, however, multiagent systems consist of institutional, human and artificial agents, and interaction processes in such mixed systems are only semi-automated, as in an enterprise where employees ('human agents') interact with organizational units ('institutional agents'), with agentified information systems ('software agents') and with agentified machines ('robots'). In any case, agents have to interact with each other in a flexible but coherent manner.

Examples of incoherent interactions in multiagent systems are:

- mis-understanding the message of a team mate in a robotic soccer game, or not giving way to another AGV although it has communicated a higher order priority;
- not making any compensation offer when being unable to deliver an item after having acknowledged the purchase order for that item, or sending a payment reminder although the payment has already been made, in an electronic commerce supply chain;
- or cancelling an appointment with a higher priority in favour of an appointment with a lower priority in agent-mediated group scheduling.

Incoherence may be caused by plain human user errors, by programming errors, by miscommunication, and by unforeseen changes in the environment of an agent. While coherence is a real concern in all types of interaction processes, it seems of particular importance for inter-organizational processes that are automated or semi-automated by means of information technologies.

The coherence problem in multiagent systems covers a larger domain than the sub-problem we try to solve in the first step of our research. On a higher level the coherence of (semi-) automated interaction processes consists of the following subproblems:

1. Enforcing *linguistic coherence*: avoiding miscommunication by speaking the same language and sharing a common understanding of its terms. This requires to use
   - a standard communication language (such as FIPA-ACL or ebXML) that defines message types and their communication semantics,
   - standard composition languages for expressing composite propositions and actions, and
   - shared ontologies (corresponding to domain models) that define the basic vocabularies used for denoting entities and their properties and relationships.
2. Enforcing the *normative coherence* of interaction processes as regulated by commitments, claims, and norms (where norms can be conceptualized as meta-commitments, i.e., as commitments referring to other commitments, see [Sin99]). The operational semantics of commitments includes procedures that define how they are created and discharged, under which (exceptional) circumstances they can be cancelled, and what happens if they are violated. Norms and commitments constrain interaction processes. Unlike integrity constraints, however, they may be cancelled and may be violated without crashing the system.
3. Enforcing *process integrity* by
   - verifying the correctness of interaction protocols (based on safety and progress properties) at design time, and by
   - checking interaction constraints and handling exceptions at runtime.

Current and forthcoming XML-based standards for electronic business communication, such as the Electronic Business XML (ebXML) initiative by UN/CEFACT and OASIS, do not address the

coherence problem in general, but are focused on limited forms of linguistic coherence. Only ebXML defines that a 'Collaboration Protocol Agreement' must exist between two parties in order for them to engage in automated interactions. Since the ebXML Business Process Specification Schema, currently still under development, is based on the rather limited and implementation-oriented definition of events and actions in UML, one may expect that the forthcoming ebXML concept of interaction protocols will not be able to capture the semantics of (semi-)automated interactions, including a proper treatment of the issues of normative coherence and process integrity.

In order to attack the problem of process integrity for fully automated interaction processes (where no human agents are involved), we would like to investigate the possibilities to translate interaction rules into Petri nets and use the well-established process verification techniques of Petri nets. The problem to be solved is how to translate a rule set into a Petri Net that preserves its operational semantics.

4. Actions and Events, Commitments and Claims

In a business domain, there are various types of actions performed by agents, and there are various types of global state changes, including the progression of time (which is perceived usually as a discrete value). For an external observer, both actions of business agents and environmental state changes constitute events. In the internal perspective of an agent that acts in the business domain, the actions of the other agents count as events but not the actions of the agent itself. In the external perspective, actions create events, but not all events are created by actions. Those events that are created by actions, such as delivering a product to the customer, are called *action events*. Examples of business events that are not created by actions are: the fall of a particular stock value below a certain threshold, the sinking of a ship in a storm, etc. In our approach, we take into consideration only the action events.

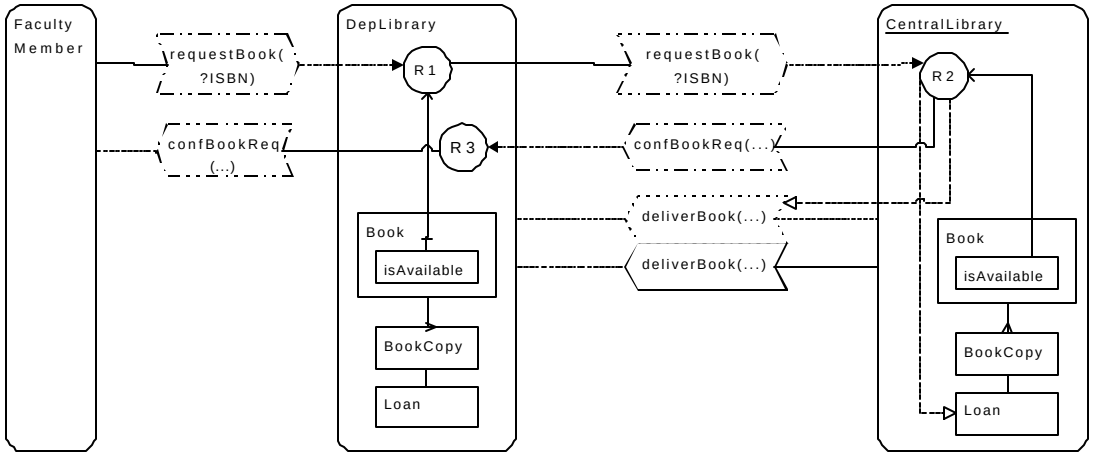


Figure 1: An interaction pattern diagram describing the process type where faculty members request books from a department library, and where the request is forwarded to the central library, since the books are not available.

We also make a distinction between *communicative* and *non-communicative* (e.g. physical) actions and events. Many typical business events, such as receiving a purchase order or a sales quotation, are communication events. Business communication may be viewed as asynchronous point-to-point message passing. The expressions receiving a message and sending a message may be

considered to be synonyms of perceiving a communication event and performing a communication act. This is different from the low level concept of method invocation in object oriented programming, because agent-oriented programming (AOP) assumes the high level semantics of speech-act-based Agent Communication Languages (ACL) messages [Wag00a].

The second fundamental concept in business interaction processes is the *commitment* concept. Representing and processing commitments and claims in information systems explicitly helps to achieve coherent behaviour in real-life interaction processes. In [Sin99], the social dimension of coherent behaviour is emphasised, and commitments are treated as ternary relationships between two agents and a "context group" they belong to. For simplicity, we treat commitments as binary relationships between two agents. Commitments to perform certain actions (or to check if certain conditions hold) arise from specific communication acts. For instance, sending a sales quotation to a customer commits the vendor to reserve adequate stock of the quoted items (for some time). Likewise, acknowledging a sales order implies the creation of a commitment to deliver the ordered items on or before the specified delivery time.

These two concepts are depending on the perspective chosen. In the perspective of a particular agent, *actions* of other agents are viewed as *events*, and *commitments* of other agents may be viewed as *claims* against them. In the internal perspective of an agent, a commitment refers to a specific action to be performed when necessary for another agent (due to time for example), while a claim refers to a specific event that must be created by an action of another agent, and has to occur in the future. In the view of an external observer, actions are also events, and commitments are also claims, exactly like two sides of the same coin. This means that in any model, a *commitment* of agent a1 towards agent a2 to perform an action r, can also be viewed as a *claim* of a2 against a1 that an action r will be performed. There is no similar coupling between actions and events, because an agent can sense an event, but due to its internal status, it may, or may not take any consequent action. It is simpler to model claims and commitments in a Petri Net, due to their mirrored nature (these are created and dismissed in pairs) and the related semantic is usually very clear and simple to represent.

5. Reaction Rules

In the *Agent-Object-Relationship* (AOR) modeling language proposed in [Wag00a,Wag00b], reaction rules may be visualized in *Interaction Pattern Diagrams* including action events and claim/commitments. An example of such a diagram is shown in Figure 1 where reaction rules are visualized as a circle with incoming and outgoing arrows drawn within the agent rectangle whose reaction pattern it represents. Each reaction rule has exactly one incoming arrow with a solid arrowhead: it represents the triggering event condition which is also responsible for instantiating the reaction rule (binding its variables to certain values). In addition, there may be ordinary incoming arrows representing state conditions (referring to corresponding instances of other entity types). There are two kinds of outgoing arrows. An outgoing arrow with a dashed line denotes a mental effect (i.e. a change of beliefs and/or commitments). An outgoing connector to an action event type denotes the performance of an action of that type. For instance, the reaction rule R2 in Figure 1 reads as follows: *When the central library receives a certain book request from a department library, it checks if a copy of that book is available, and if this is the case, the request is confirmed, and a new corresponding loan object as well as a commitment to deliver the requested book in due time is created.*

A reaction rule consists of an antecedent expression (its 'body') and a consequent expression (its 'head'). The antecedent is composed of two parts: an event condition (where an event is represented

by an incoming message), and a state condition (to be checked against the state of the agent that is executing the rule). Also, the consequent is composed of two parts: an action term and an effect formula that specifies the state changes effected by the action performance. We use a tabular notation for specifying reaction rules, such as in Table 1.

ON	**Event**	*book request from a department library*
		`RECEIVE` requestBook(?ISBN) `FROM` ?DepLib
IF	**Condition**	*a copy of that book is available* BookCopy.isAvailable(?ISBN, ?InvNo)
THEN		
	Action	*confirm the request*
		`SEND` confBookReq(?ISBN)
	Effect	*a new corresponding loan object as well as a commitment to deliver the requested book in due time is created*
		`CREATE COMMITMENT TOWARDS` ?DepLib `TO` deliverBook(?ISBN) `BY` tomorrow(); `CREATE BELIEF` Loan(?DepLib, ?ISBN, ?InvNo, today()

Table 1: The interaction rule R2 of Figure 1 in tabular form.

6. Translating Reaction Rules into Petri Net Transitions

Since reaction rules specify state transitions, it seems natural to relate them to Petri net transitions. The state of an e-business scenario consists of the private state components of each agent (such as its privately held beliefs) and of the public state components which are shared among two or more agents, such as the current action events and commitments/claims. We are interested in those cases where the state of a scenario can be represented by means of a certain token population of the places of a (preferably classical) Petri net, and the behavior (i.e. the reaction rules) of the involved agents can be represented by means of transitions in this net.

In order to obtain classical Petri nets, we have to make some severe restrictions:

- the state condition consists of a single *propositional variable* (or a conjunction of propositional variables)
- the event consists of an incoming *message without content parameters*
- the action consists of one or more outgoing messages without content parameters
- the effect consists of a single propositional variable (or a conjunction of propositional variables)

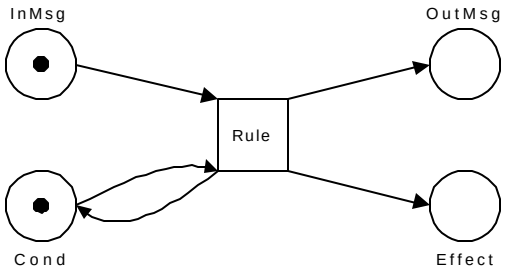


Figure 2

Under these assumptions we obtain the PN shown in Figure 2, where `InMsg` and `OutMsg` are places that represent message buffers, and `Cond` and `Effect` are places that represent propositions. Notice that, while a message is consumed and can therefore be represented as a token on an ordinary place, a state condition is normally not 'consumed' but rather preserved after firing the transition. The preservation of a condition can be achieved by a reproduction arc as in Figure 2, although this is not a very natural representation of what is really going on.

Since both a condition and an effect formula may be a negated proposition, we have to introduce *negative* input and output arcs as in Figure 3.

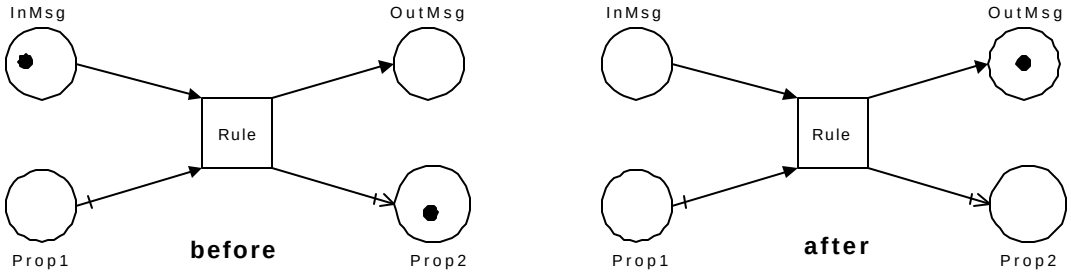


Figure 3: Firing a transition representing a reaction rule with a negative condition and a negative effect.

Thus, if the behavior of an agent is specified by a set of reaction rules satisfying the above restrictions, we can translate it into a Petri net whose transitions may be only linked through the propositional places (if we do not consider the technical possibility to send a message to oneself).

7. Multiagent Nets

For representing a complete process type involving several agent types, such as in Figure 1, we may translate the rule set of each involved agent type into a single agent Petri net such that these nets are linked to each other via the shared message buffer places. Notice that the single agent Petri nets must not be linked via private propositional places because they can be only accessed by their owner. We call the resulting type of Petri net a *multiagent net*.

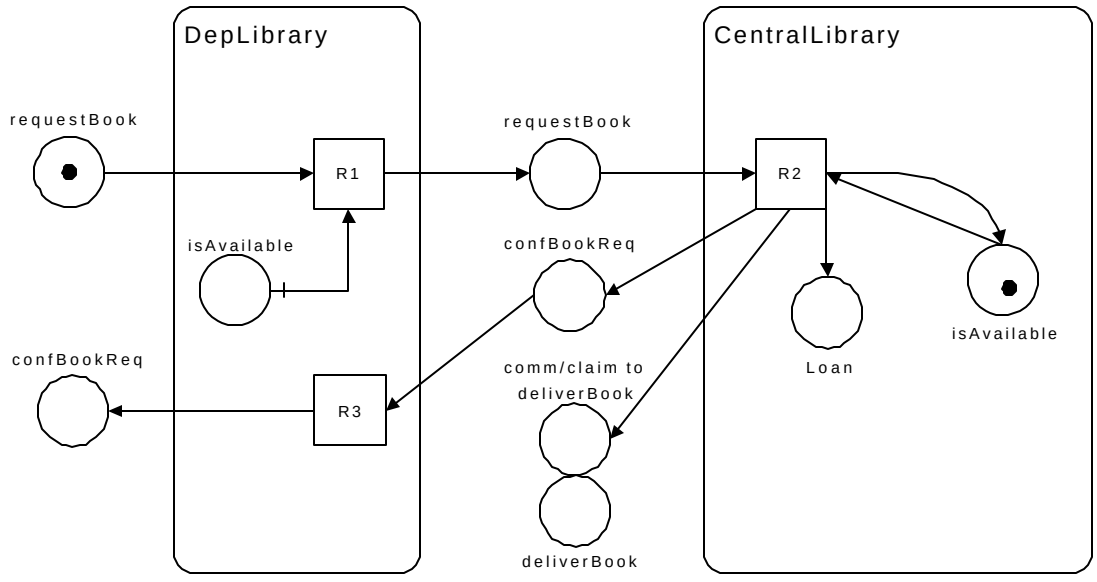


Figure 4: A multiagent net corresponding to the interaction pattern diagram of Figure 1.

The multiagent net shown in Figure 4 corresponds to the interaction pattern diagram of Figure 1. The two single agent nets are drawn in a rectangle with rounded corners. They contain the rules and the private beliefs of the respective agent. Both messages and commitments/claims are shared between two or more agents, so they are drawn outside of any agent rectangle.

Since we only have one type of place in the multiagent net, it is visually less clear than the interaction pattern diagram. However, we hope that by translating reaction rules into Petri nets, we will be able to benefit from the formal analysis techniques that have been established for them.

9. Conclusions and future work

This research is still in a preliminary stage. We presented in this paper some techniques how to translate an interaction process type specified by reaction rules into a Petri net. To perform experiment with agents we are using the PROVE system (see [Szi00] and [Aer00]) that bis based on the production rule engine JESS (Java Expert System Shell). The syntax of Jess rules is almost similar with CLIPS, and the rules described in the AOR diagrams must be translated (by a rule generator linked to the AOR enabled-editor) to obtain these in the format needed by the inference engine used by the agents. In this paper, for the sake of simplicity, we are using a tabular (schema-like) description of the rules. CLIPS code, based on predicate logic, is too cluttered to be easily readable.

It would be nice to include verification in a more general agent-oriented development method, where the system and the interaction processes are designed using the AOR modeling language, and the rules can be generated automatically in CLIPS format and also translated from the AOR description into Petri Nets. Probably, the second translation will be semi-automated, needing human intervention to define new predicates for linking the rules in the sequences dictated by the interaction process structure.

Another conclusion is that any multiagent system which can be described in the AOR modeling language, where the rules have a more generic structure, can be verified in this way. The problem is that the AOR specification of the system could take time, and it is probably simpler to translate the interaction process directly into Petri Nets. We still have to investigate which way is the most straightforward.

The next step in our research is to try to fully automate the translation process. Conversely, we want to know which are the best ways to specify the rules to make them easily translatable. One of the first guidelines is to try to program in a way which enables the chaining of rules. This is happening however in any rule-based system which is carrying out a process, but the condition/places are not explicitly specified.

References

[Aal00] Aalst, W.M.P.,vd, (2000), "Loosely Coupled Interorganisational Workflows: Modelling and Analysing Workflows Crossing Organisational Boundaries", Information and Management, vol. 32, no. 2, March 2000, pp.67-75.

[Aal01] Aalst, W.M.P., vd, (2001), "The P2P Approach to Interorganisational Workflows", Proc. of 13th Intl. Conference on Advanced Information Systems Engineering (CAiSE'01), Springer, Berlin.

[Aer00] Aerts, A.T.M., Szirbik, N.B., Hammer, D.K., Goossenaerts, J.B.M., Wortmann, J.C., (2000), "On the Design of a Mobile Agent Web for Supporting Virtual Enterprises", WETICE2000 Conference, Gaithersburg, NY.

[Bas98] Basten, T., (1998), In Terms of Nets: System Design with Petri Nets and Process Algebra, PhD thesis, Eindhoven University of Technology, December 1998, Eindhoven, The Netherlands.

[Naz93] Nazareth, D.L., (1993), "Investigating the applicability of Petri Nets for Rule-Based System Verification", IEEE Trans. on Knowledge and Data Engineering, vol5, no. 3, June 1993, pp.402-416.

[Sin99] Singh, M.P., (1999), "An Ontology for Commitments in Multi-Agent Systems", Artificial Intelligence and Law, no. 7, July 1999, pp.97-113.

[Szi00] Szirbik, N.B., Wortmann, J.C., Hammer, D.K., Goossenaerts, J.B.M, Aerts, A.T.M., (2000), "Mediating Negotiations in a Virtual Enterprise via Mobile Agents", AIWORC2000, Buffalo, NY.

[Wag98] Wagner, G., (1998), Foundations of Knowledge Systems with Applications to Databases and Agents, Kluwer Academic Publishers.

[Wag00a] Wagner, G., (2000), "Agent-Oriented Analysis and Design of the Organizational Information Systems", Proc. of Fourth IEEE Intl. Workshop on Databases and Information Systems, May 2000, Vilnius, Lithuania.

[Wag00b] Wagner, G., (2000), "The Agent-Object-Relationship Meta-Model: Towards a Unified Conceptual View of State and Dynamics", Technical Report, Eindhoven Univ. of Technology, October 2000, Eindhoven,
http://tmitwww.tm.tue.nl/staff/gwagner/AOR.pdf

Application of Petri Nets in Modelling Distributed Software Systems – Abstract –

Guido Wirtz
Distributed Systems Group, Computer Science Institute,
Westfälische Wilhelms-Universität, Einsteinstraße 62, 48149 Münster, GERMANY
guidow@math.uni-muenster.de

There is a fairly long tradition in suggesting that Petri nets and their underlying concepts are a suitable tool for different phases of the software development process. The emphasis, clearly, has been put on obtaining adequate system models during the design phase. Taking the users view, the benefits mentioned usually, are the simple, easy to understand visual notation combined with the expressiveness of modelling nondeterminism as well as concurrency, and the intuitive semantics of playing the token game, i.e., the model is operational and can be evaluated through simulation. Looking at the theoretical background, there is a clear syntax and semantics for the graphical as well as other related formalisms, which – besides simulation – allows for the analysis of nets and their transformation into executable code.

Taking a view on the state-of-the-art in the software engineering community and the formalisms and languages used, we see a completely different picture. There are many other notations and formalisms in high and frequent use, e.g., the Unified Modelling Language with various notations, thanks to Harel's statecharts at least one formalism among them. Petri-Nets usually find their applications in some rather specific areas like, e.g., telecommunication protocols.

So, what went wrong with Petri-Nets in software engineering ? First of all, a lot of steps during software development have their own traditional notations. For describing the static aspects of software like, e.g., modules and their dependencies or entity-relation-like structure diagrams, Petri-Nets may not be suitable at all. Second, software engineers often neglect the benefits of Petri-Nets related to concurrency by totally ignoring concurrency and related issues. The primary reason for the situation, however, is the mismatch between the folklore about Petri-Net benefits and the problem that the expressiveness of basic Petri net formalisms, i.e., place-transition nets and variants, is insufficient to handle many real-life modelling problems because of their mere size. Although there are extensions towards high-level Petri-Nets around for more than 20 years now and methods for modularization or refinement while preserving properties of the parts are under investigation for nearly as long, the resulting net formalisms tend to lose their benefit of being easy to understand frequently. Developments in software engineering, in particular the introduction of object oriented concepts have added their share to the problem. Adapting Petri-Net formalisms to object-orientation does not make the formalism any easier and often the problems users experience with object-oriented Petri-Nets are problems with object-orientation altogether.

Nevertheless, there is a chance to overcome some of these problems and make Petri-Nets a successful candidate for several steps and aspects of software engineering where they are suited for. (There is, however, no need to draw structure diagrams with Petri-Nets.) At least, this holds in the context of distributed software systems for several good reasons.

In distributed systems software projects, concurrency is a topic and can not be ignored without jeopardizing the success of the entire project. So, the unique potential of Petri-Nets for describing concurrency, resource scheduling, coordination of conflicting goals and so on is of practical use. Moreover, distributed software systems, which are often close to standard middleware layers, are accepted as being of a high complexity that cannot be hidden from the developer. In contrast, the explicit government of these problems in detail is required to obtain useful systems with a guaranteed behaviour.

In distributed systems, structuring mechanisms for breaking complexity are as needed as in standard software. Unfortunately, the classical notions of abstraction and interfaces are fixed to static aspects like names, parameter types and so on. Interacting parts of a distributed system are more concerned about behaviour, e.g., availability of requested services, the chance for being blocked for some time or even permanently. Here, Petri-Nets and their capabilities for analysis can be of great help for designing protocols with behaviour. The coordination of resources in a distributed implementation may also be described using Petri-Nets and the possible effects can be investigated for, e.g., proving the behaviour protocols correct under specific resource assumptions.

There are more possibilities for applying Petri-Nets during design. When using agents as a metaphor to structure systems during software development, to give one more example, the behaviour of agents, the different context situations provided for agents as well as the effects of interaction between different agents or with the context on the state of an agent or the context can be described by Petri-Nets.

For nowadays object-oriented software development processes, there are a number of challenges to be solved before Petri-Nets have the chance to gain the level of general acceptance in the modelling community that matches their potential benefits:

- Identify specific aspects in the different steps of object-oriented software development where the application of Petri-Nets can be a benefit and support these aspects by adequate Petri-Net languages and tools.
- Find methods to apply *families of Petri-Net variants* in a sequence ranging from their informal use to a level of detail that allows for strong analysis and/or code generation without losing too much information, e.g., analysis results, through the different steps. At least, there should be a (coarse) notion of consistency between the different levels of detail.
- Identify methods to solve the well-known issues of modularity and scalability in a manner that fits into the software development process and remains understandable to software engineers.
- How much of the underlying semantics of object-oriented high-level Petri-Nets and related models have to be visible to (can be hidden from) the user without making the formalism unusable at all ? At the moment, supporting a sufficient amount of hiding seems to be the more severe problem.

These goals are centered around the question how to fit Petri-Nets into the used software development process rather than tayloring the process around Petri-Nets. There is no realistic chance to achieve the latter. The manner in which some of the challenges should be tackled, seem to depend on the needs of the application domain at hand, and so we will have different suitable Petri-Net-based formalisms and notations for different areas rather than a single fit-them-all approach.

Taking one more step, the applicability of the modelling power of Petri-Nets is not restricted to software engineering for distributed systems. Component-based complex sequential systems may need as much coordination, interfaces with behaviour descriptions and possibilities to check their potential interaction patterns as distributed systems. Under many circumstances, agent-oriented modelling may be a good idea to obtain flexible models of sequential software. Moreover, the entire range of high-level modelling, namely *business process modelling* and *workflow modelling* are promising candidates. Because the needs for modelling concurrency and ressource scheduling have been recognized in these areas for years, even today the acceptance of Petri-Nets is much higher there than in the software community.

Note: a version of the full paper can be obtained as a Technical Report from the author's homepage (URL: `http://wwwmath.uni-muenster.de/cs/u/versys/index.html`).