Object Coordination Nets 2.0
Semantics Specification

Holger Giese

FB 15, Institut für Informatik, Westfälische Wilhelms-Universität
Einsteinstr. 62, D-48149 Münster, Germany

email: gieseh@math.uni-muenster.de

Bericht Nr. 15/99 - I
Due to hardware developments, strong application needs and the overwhelming influence of the net in almost all areas, distributed systems, especially software systems, have become one of the most important topics for nowadays software industry. Unfortunately, distribution adds its share to the problems of developing complex software systems. Heterogeneity in both, hardware and software, concurrency, distribution of components and the need for interoperability between different systems complicate matters. Although subject to permanent changes, distributed systems have high requirements w.r.t. dependability and performance. Moreover, new technical aspects like resource management, load balancing and deadlock handling put an additional burden onto the developer.

Nowadays programming languages are able to deal with these problems but work on a level of abstraction which is much too low. This makes complex systems hard to implement and much harder to maintain and change. Nowadays design methods fail to describe the specific properties of distributed systems in an adequate manner. One of the essential drawbacks of the methods which are around for some time now, is the lack of support for the dynamic aspects of distributed software systems.

Based on object-oriented analysis and design, our approach adds the aspects of architectural modeling to the usual structural modeling of distributed software systems. Moreover, behavioral modeling w.r.t. concurrency, competition for resources and coordination is made more intuitive by means of an adequate modeling language. We use a specific kind of high-level petri-nets, so-called Object Coordination Nets (OCoNs). The result is an integrated design language which seamlessly combines the well-known approaches of object-oriented analysis and development with methods which work well for the specific aspects of distributed software systems. Our long term goal is an integrated development methodology for distributed software systems. Currently, we are working on a sound theoretical framework for contract-based reasoning, advanced tool support for modeling, analysis and code generation as well as non-toy case studies to evaluate our approach.

More information concerning the OCoN project as well as the Distributed Systems Group at the Institute for Computer Science at Münster University are available under http://wwwmath.uni-muenster.de/cs/u/ocon/index.html.

Guido Wirtz
Münster, March 1999
Abstract For distributed object or component systems, a suitable software architecture and a strong separation of modules is necessary. Current visual notations have several drawbacks: concurrency support is very limited and they fail to integrate the external state based view of objects when aspects of data and control flow are specified. Hence, they are not sufficient to support a seamless contract based design style.

The report first defines a modular Petri net formalism named coordination nets that supports a port concept inspired by the $\pi$-Calculus. The level 2 conformance with the forthcoming ISO high level Petri net standard is demonstrated in the appendix.

Based on this flexible net dialect, the Object Coordination Net or short OCoN formalism for the specification of behavior aspects in an object-oriented distributed system is defined. It distinguishes several distinct net diagrams. The formalism allows mixed event and state based true concurrent modeling. It describes contracts, object scheduling, resource handling and the abstract data and control flow of services. A seamless integration of contract specifications into service and object scheduling specifications is provided. Although abstract, the OCoN formalism remains operational which permits abstract simulation and guarantees a feasible implementation.

Keywords: object-oriented design, behavior specification, contracts, distributed systems, visual design notations, Petri-Nets, OCoNs
Contents

1 Introduction
   1.1 Aims and Related Work ............................ 5
   1.2 Structure of the Report ............................. 5

2 Petri Nets
   2.1 Place Transition Nets .............................. 7
       2.1.1 Basic Definitions ............................ 7
       2.1.2 Basic Net Definitions .......................... 8
       2.1.3 Non Concurrent Step Semantic .................... 9
       2.1.4 Trace and Failure Sets ......................... 10
       2.1.5 Concurrent Step Semantic ....................... 12
       2.1.6 Runs ................................... 12
       2.1.7 Progress and Fairness .......................... 14
   2.2 High Level Petri Nets .............................. 15
       2.2.1 From Place Transition Nets to High Level Petri Nets ........ 16
       2.2.2 High Level Petri Net Graphs ...................... 17
   2.3 Extensions for Modularity ........................... 19
       2.3.1 Object-Oriented Nets .......................... 20
   2.4 Extensions for Interaction ........................... 21
       2.4.1 Asynchronous and Synchronous Communication/FIFO ....... 21
       2.4.2 Procedure Calls ............................. 21
       2.4.3 Port Passing Calculi .......................... 22
   2.5 Design Decisions ................................ 22

3 Coordination Nets 23
   3.1 Basic Concept .................................. 23
       3.1.1 Requirements ................................ 23
       3.1.2 Architecture ................................ 23
       3.1.3 Communication ............................... 24
       3.1.4 Create .................................. 25
       3.1.5 Example . .................................. 25
   3.2 Typing Concept ................................. 27
       3.2.1 Literal Typing ................................ 27
       3.2.2 Port Typing ................................ 27
       3.2.3 Remote Procedure Call with Asynchronous Reply .......... 32
       3.2.4 Sharing a Provided Port ...................... 33
       3.2.5 Actions and Abstraction ...................... 33
       3.2.6 Conformance ................................ 35
   3.3 Coordination Nets ................................ 36
       3.3.1 Types .................................. 37
       3.3.2 Port Typing ................................ 39
3.3.3 Port Typing ........................................... 41
3.3.4 Typing Places and Object Orientation ................ 42
3.3.5 Annotations .......................................... 43
3.3.6 Coordination Net Graph ................................ 45
3.3.7 Syntactical Restrictions ................................ 46
3.3.8 Coordination Net System ................................ 48

3.4 Non Concurrent Step Semantic .......................... 49
3.4.1 Valid Assignments and Guards ....................... 50
3.4.2 Pre-Conditions ....................................... 50
3.4.3 Enabling .............................................. 52
3.4.4 Post-Conditions ....................................... 52
3.4.5 Reverse .............................................. 54
3.4.6 Transition Rule ....................................... 54

3.5 Concurrent Step Semantic ................................ 54
3.5.1 Pre Conditions ........................................ 55
3.5.2 Enabling .............................................. 55
3.5.3 Post Conditions ....................................... 55
3.5.4 Transition Rule ....................................... 56
3.5.5 Runs .................................................. 56

3.6 Fairness and Progress ................................... 58

4 Extension with Actions and Pools ......................... 59
4.1 Actions ................................................ 59
4.1.1 Calls .............................................. 61
4.1.2 One Way Call ........................................ 64
4.1.3 Net Creation and Invocation ......................... 64
4.1.4 Call Forward ........................................ 64

4.2 Pools .................................................. 65
4.2.1 Event Pools ........................................ 66
4.2.2 Resource Pools ....................................... 66
4.2.3 Locking and Releasing Resources .................... 69
4.2.4 Lock Semantic ....................................... 70

4.3 Selection and Replication ................................ 71
4.3.1 Complex Action Scheme ................................ 71
4.3.2 Usual Edge .......................................... 74
4.3.3 Selection Edge ....................................... 74
4.3.4 Parallel Replication Edge ............................ 75
4.3.5 Sequential Replication Edge ......................... 76
4.3.6 Example .............................................. 76

4.4 Net Creation ........................................... 77
4.4.1 Single Task Net ...................................... 77
4.4.2 Creation of Subsystems ................................ 78

4.5 Conventions ............................................ 79
1 Introduction

This report describes the formal basis for the object coordination net approach introduced by Giese, Graf and Wirtz in [WGG97, GGW98, GGW99c, Gie99, WGG99]. The OCoN formalism and its graphical notation are designed for the specification of behavioral aspects for distributed object systems.

1.1 Aims and Related Work

An essential goal of our approach is to allow to model concurrency as natural as possible. The approach is considered for design and prototyping of distributed software systems and thus we assume transparency concerning heterogeneity and distribution.

In our opinion, object-orientation and object-oriented design are the most promising approaches for distributed system modeling, because the combination of functionality and data abstraction leads to a possible distribution of system entities. But not every object-oriented design style really supports the modularization and is sufficient for distributed systems. Traditional approaches for behavior modeling of object-oriented systems (see Beringer [Ber97] for a good overview) are not sufficient to describe systems that contain concurrency.

By using the basic software engineering ideas of modularization and separation applying the ”design by contract” idea of Meyer [Mey94, Mey97] and ”separation of concern” like Wirfs-Brock [WBWW90], a suitable design approach is feasible. We further restrict our focus on the coordination aspect in the sense of Wegner [Weg96, Weg97] and assume a design style that separates different views on an object like role base modeling [Ree96]. Our approach is based on contracts containing interfaces with protocols that can be considered to be architectural connectors characterized by Allen and Garlan in [AG97]. During the design we represent the structure of the software architecture [SG96] by objects and their relations. These objects are then responsible for the coordination of the contained and related other elements.

We base our approach on the upcoming standard for object-oriented modeling, the UML [Rat97]. It provides a set of extension mechanisms. We use these mechanisms to incorporate our notations.

1.2 Structure of the Report

We start our considerations with some basic definitions and a definition of place transition nets described in the terms of the high level Petri net standard in section 2. The basic formalisms and concepts of the high level Petri net standard are reconsidered in section 2.2. Also several high level Petri net approaches and extensions are discussed. Our basic formalism to model system behavior in a visual manner is a Petri net formalism that
is presented in section 3. An already object-oriented type system for ports is provided there. Some useful higher level behavioral elements are presented in the following section 4 building the basis for our final object-oriented notion. The formalism is extended to specify object-oriented systems based on a class notion using the object coordination net formalism in section 5. In the appendix A, the level 2 conformance with the ISO high level Petri net standard is demonstrated.

Reader familiar with high level Petri net concepts may skip section 2. To informally understand the OCoN semantic it is sufficient to read the basic concept and typing concept of section 3.1 and 3.2. The formal syntax and semantic of the underlying coordination net formalism described in detail in section 3.3 to 3.6 may then be skipped. Section 4 provides the definition of several graphical abstractions and their definition in terms of the coordination net formalism which finally form the object coordination net formalism presented in section 5.
2 Petri Nets

Petri nets [BRRe87, Rei85] are a well founded formal method to describe the behavior of a system with the background model of partial order runs. An intuitive visual notation like for state machine exist. To ease the understanding we will first define the semantic and structure for untyped place transition nets with the terms and conventions of the high level Petri net standard. Readers with appropriate background may skip this and the next subsections about high level Petri nets and continue with section 3, where the basic concept for the coordination net formalism is presented. Some basic definitions of the mathematical framework are presented in section 2.1.1. The definitions of the coordination nets in subsection 3.3 follows the scheme here presented.

2.1 Place Transition Nets

We will consider place transition nets as the standard class of Petri nets as well as the restricted variants occurrence nets or state machine nets to introduce the basic concepts for net based semantic and specification.

2.1.1 Basic Definitions

The basic notation of a set with elements $e_1,e_2,\ldots$ is given by $\{e_1,e_2,\ldots\}$. We usually use capital letters or words to denote sets. Based on two sets $M$ and $M'$ a function $f$ can be defined. We can further distinguish a total function $M \mapsto M'$, where for each $m \in M$ $f(m)$ is defined and element of $M'$, from a partial function $M \leadsto M'$, where $f(m)$ may be undefined for a certain $m \in M$. A function is named injective iff $\forall m,m' \in M \quad f(m) = f(m') \Rightarrow m = m'$ and named surjective iff $\forall m' \in M' \exists m \in M \quad m' = f(m)$. The class of all subsets of a set $M$ is denoted as $\wp M$, the power-set where for all $M' \subseteq M$ holds $M' \subseteq M$. An alternative representation is a characteristic function $M' : M \mapsto \text{Bool}$, which results in true for every element of $M$ also in $M'$ and false otherwise. This representation using a total function on $M$ to $\text{Bool}$ can be generalized to also allow multiple occurrences of the same element by mapping a set to $\mathbb{N}$. A multi-set of a set $M$ is such a total function $M \mapsto \mathbb{N}$ and $\mu M$ is the set of all multi-sets over $M$. Often, for a multi-set $B \in \mu M$ the symbolic sum representation

$$B := \sum_{m \in M} B(m) \ m$$

can be used and the multi-set inclusion $\leq$, multi-set sum $+$ and multi-set subtraction $-$ can be defined by

$$B \leq B' \iff \forall m \in M \quad B(m) \leq B'(m),$$

$$B + B' := \sum_{m \in M} (B(m) + B'(m)) \ m \quad \text{and}$$

$$B - B' := \sum_{m \in M} (B(m) - B'(m)) \ m,$$
where $B - B'$ is only well defined, if $B' \leq B$ holds. If a multi-set $B \in \mu M$ is also a set $(\forall m \in M \ B(m) \leq 1)$ we simply write $B \subseteq M$ or $B \in \varnothing M$. We use sets of pairs $M \times \mathbb{N}$ of the form $(a, B(a))$ for every $a \in M$ with $B(a) \geq 1$ to describe a $B \in \mu M$ and can thus write for example $B = \{(a, 3), (b, 1)\}$ for a multi-set over $M = \{a, b, c, d\}$. To extend a multi-set on the cross product $M \times M'$ of the sets $M$ and $M'$ we define the multi-set cross product extension for an element $m' \in M'$ and a multi-set $B \in \mu M$ by

$$m' \times B := \sum_{m \in M} B(m)(m', m) \in \mu (M' \times M).$$

A word over a set $M$ is an ordered sequence $a_1 a_2 a_3 \ldots$ of not necessarily different elements $a_i \in M$ and the set of all finite words over $M$ is denoted by $M^*$. Corresponding definitions for the prefix containment $\preceq$, the append operator $;$ and the prefix cut $-$ are

$$w \preceq w' \iff \exists v \in M^* \ wv = w',$$

$$w; w' := ww' \quad \text{and}$$

$$w - w' := v \ (\exists v \in M^* \ w'v = w),$$

where also $w - w'$ is only well defined if $w' \leq w$. In contrast to the multi-set $+$, the $;$ operation on words is not commutative and thus $w; w'; w'' \neq w; w''; w'$ in general. On words we additionally assume a symbol for the neutral element concerning the append $\epsilon$ that represents the empty word. It holds $w; \epsilon = \epsilon; w = w$ for any $w \in M \cup \{\epsilon\}$.

### 2.1.2 Basic Net Definitions

The basic Petri net is simply a bipartite graph described by the following definition.

**Definition 2.1** A triple $N = (S, T, F)$ is a Petri net or net graph, iff

- $S$ a set of nodes, called places,
- $T$ a set of nodes, called transitions, disjoint from $S$ ($S \cap T = \emptyset$), and
- $F \subseteq (S \times T) \cup (T \times S)$ a set of arcs, known as the flow relation.

We define $\bullet x = \{y \in S \cup T | (y, x) \in F\}$ and $x \bullet = \{y \in S \cup T | (x, y) \in F\}$.

For $S$ and $T$ finite we call the Petri net a finite Petri net\(^1\). By adding a marking and edge weight a place transition net can be build.

**Definition 2.2** A place transition net is a 5-tuple $PTN = (S, T, F, w, M)$ with

\(^1\)usually Petri nets are finite ones, but for infinite runs we need this more general definition (see [Rei98])
2.1 Place Transition Nets

\((S, T, F)\) a Petri net, 

\(M : S \mapsto \mathbb{N}\) a marking \((M \in \mu S)\) and 

\(w : F \mapsto \mathbb{N}\) an edge weight for \(F\).

Like the high level Petri net standard we will use the names \texttt{PLACE} for \(S\) and \texttt{TRANS} for \(T\) to identify the set of places and transitions. This direct correspondence for the place and transition sets will not remain when high level Petri nets are considered. The needed generalization is demonstrated in subsection 2.2.1.

![Figure 1: A Petri net and a corresponding place transition net](image)

An example for a Petri net and its extension to a place transition net is given in figure 1. The edge weight is given by annotated natural numbers (1 as standard weight may be omitted) to each edge and the marking is specified by tokens in each place visualized as black dots. The net has the initial marking \(M_0 = \langle s_0, 1 \rangle, \langle s_1, 2 \rangle\).

2.1.3 Non Concurrent Step Semantic

The \texttt{pre-condition} for a single transition \(t \in \text{TRANS}\) is defined as 

\[\text{Pre}(t) := \sum_{(s,t) \in F} w((s, t)) s \in \mu \text{PLACE}\]

with \(\mu \text{PLACE}\) is the set of all total functions \(S \mapsto \mathbb{N}\) and thus \(t\) is \textit{enabled} for a marking \(M \in \mu \text{PLACE}\) iff 

\[\text{Pre}(t) \leq M \equiv \forall s \in S \text{ Pre}(t)(s) \leq M(s),\]

where \(\text{Pre}(t)\) is in \(S \mapsto \mathbb{N}\) and thus \(\text{Pre}(t)(s) \in \mathbb{N}\). The \texttt{post-condition} is analogous defined as 

\[\text{Post}(t) := \sum_{(t, s) \in F} w((t, s)) s \in \mu \text{PLACE}\]

and \(t\) can \textit{occur} for a marking \(M\) and transform the system state to marking \(M'\) if \(t\) is \textit{enabled} for \(M\) as follows 

\[M' := M - \text{Pre}(t) + \text{Post}(t)\]
We write \( M[t]M' \) or \( M \xrightarrow{t} M' \). We extend this notion to sequences or *traces* of occurrences given by word \( t_1; \ldots ; t_n \in \text{TRANS}^* \) by defining \( M[t_1; \ldots ; t_n]M' \) iff \( \exists M_1, \ldots , M_{n-1} \) and for \( M_0 := M \) and \( M_n := M' \) holds for all \( i \in [0 : n-1] \) \( M_i[t_{i+1}]M_{i+1} \). Thus, \( M[\epsilon]M \) is contained as a special case. We write \( \{ M_0 \} \) to denote \( \{ M'[\exists t_1, \ldots , t_n M_0[t_1; \ldots ; t_n]M' \} \) the set of all reachable markings from \( M_0 \).

For the example net of figure 1 and transition \( t_2 \in T \) we have

\[
\begin{align*}
\text{Pre}(t_2) &= \langle \langle s_2, 2 \rangle, \langle s_3, 2 \rangle \rangle \\
\text{Post}(t_2) &= \langle \langle s_0, 1 \rangle, \langle s_1, 2 \rangle \rangle 
\end{align*}
\]

and the marking \( M_0 = \langle \langle s_0, 1 \rangle, \langle s_1, 2 \rangle \rangle \) is transformed to \( M_1 \) by an occurring \( t_0 \) as follows

\[
\begin{align*}
\text{Pre}(t_0) &= \langle \langle s_1, 1 \rangle \rangle \\
\text{Post}(t_0) &= \langle \langle s_2, 1 \rangle \rangle \\
M_0 = \langle \langle s_0, 1 \rangle, \langle s_1, 2 \rangle \rangle & \quad \{ t_0 \} \quad \langle \langle s_0, 1 \rangle, \langle s_1, 1 \rangle, \langle s_2, 1 \rangle \rangle = M_1.
\end{align*}
\]

### 2.1.4 Trace and Failure Sets

For an alphabet \( A \) and a labeling function \( l : \text{TRANS} \mapsto A \) mapping all occurrences of transitions \( t \in \text{TRANS} \) to labels \( a \in A \) we can consider traces and failure notions. A simple extension of \( l \) to words of transitions \( w_i \in \text{TRANS}^* \) can be defined by \( l^*(t_1; w_i) := l(t_1); l^*(w_i) \) and \( l^*(t) = l(t) \). Thus a step notion based on a labeling \( l \) can be defined by

\[
M \xrightarrow{a} M' \iff \exists t \in \text{TRANS} \quad M[t]M' \land l(t) = a
\]

and the extension to step sequences by \( M \xrightarrow{w \ast} M' \) iff \( \exists t_1; \ldots ; t_n \) and it holds

\[
M[t_1; \ldots ; t_n]M' \land l^*(t_1; \cdots ; t_n) = w.
\]

When the labeling alphabet also contains the empty word \( \epsilon \), the situation becomes more complex, because several internal step sequences extend the step notion. The resulting step notion is

\[
M \xrightarrow{a} M' \iff \exists t_1; \ldots ; t_n \quad M[t_1; \ldots ; t_n]M' \land l^*(t_1; \cdots ; t_n) = a.
\]

Cycles of \( \epsilon \) steps introduce some kind of phenomena that can be interpreted in several ways. Hoare uses the divergence notion to cover this aspect in CSP [Hoa85] and extends the failure trace model by a set of traces resulting in a divergent state. We will further assume all considered labeled place transition nets to contain no \( \epsilon \) cycles and thus can omit such special treatment.

If no special labeling is needed, the identity function on \( T \) can be used. For a marking \( M \in \mu\text{PLACE} \) the set of *initial labeled steps* init is defined by

\[
\text{init}(M) := \{ a \in A | \exists M' M \xrightarrow{a} M' \}.
\]
The possible set of traces is then defined by
\[ \text{trace}(M) \coloneqq \{ w \in A^* \mid \exists M' \xrightarrow{w} M' \} \].

Also the set of failures can be defined by
\[ \text{failure}(M) \coloneqq \{ (w, R) \mid \exists M' \xrightarrow{w} M' \land R \subseteq A \land R \cap \text{init}(M') = \emptyset \} \].

For our example of figure 1 we get
\[ \text{init}(M_0) = \{ t_0, t_1 \} \]
\[ \text{init}(M_1) = \{ t_0, t_1 \} \]
\[ \text{trace}(M_0) = \{(t_0; t_0; t_1; t_2) + (t_0; t_1; t_0; t_2) + (t_1; t_0; t_0; t_2)\}^* \]
\[ \text{failure}(M_0) = \{ (\epsilon, \{t_2\}), (t_0, \{t_2\}), (t_1, \{t_2, t_1\}), \ldots \} \],

where + and * are the usual notion for regular expressions.

The failure trace semantic distinguishes behavior only concerning the synchronization and thus for finite Petri nets with a finite set of reachable markings, a failure trace equivalent place transition net can be build that is restricted to a finite state machine.

**Definition 2.3** A place transition net \( N = (S, T, F, w, M_0) \) is a state machine net iff
\[ \forall t \in T \mid | \cdot t | = | t \cdot | = 1 \land \forall e \in F \ w(e) = 1 \land \exists s_0 \in S \ M_0 = \langle \langle s_0, 1 \rangle \rangle. \]

The corresponding non deterministic finite automaton omitting final states is given for \( M_0 = \langle \langle s_0, 1 \rangle \rangle \) by
\[ (S, \{(s, a, s') \mid \exists t \in T, (s, t), (t, s') \in F \land l(t) = a \}, s_0). \]

![Figure 2: Reachability graph](image_url)

For the general case of a place transition net we can build the graph of reachable markings and possible steps which is equivalent concerning failure trace semantics to the place transition net itself. The reachability graph presented in figure 2 describes the corresponding labeled transition system for the place transition net of figure 1 defined by
\[ ([M_0], \{(M, t, M') \mid \exists t \in T M[t] M' \}, M_0). \]
Concerning its true concurrent behavior there exist several distinct place transition nets with equivalent reachability graphs or failure trace semantic. If we want to further distinguish them, a concurrent step semantic is needed.

2.1.5 Concurrent Step Semantic

By considering a multi-set of transitions $T_\mu \in \mu$TRANS with $\text{TRANS} = T$ we can extend the formalism to the true concurrent case. By using the linear extension of the single pre-condition we get

$$\text{Pre}(T_\mu) := \sum_{t \in T_\mu} T_\mu(t) \text{Pre}(t) \in \mu \text{PLACE}.$$ 

Thus $T_\mu \in \mu$TRANS is enabled for a marking $M \in \mu \text{PLACE}$ iff

$$\text{Pre}(T_\mu) \leq M.$$ 

The post-condition is also the linear extension

$$\text{Post}(T_\mu) := \sum_{t \in T_\mu} T_\mu(t) \text{Post}(t) \in \mu \text{PLACE}$$

and the resulting marking $M'$ when $T_\mu$ occurs for marking $M$ is

$$M' := M - \text{Pre}(T_\mu) + \text{Post}(T_\mu).$$

We write $M[T_\mu]M'$ or $M \xrightarrow{T_\mu} M'$. The set of reachable markings $[M_0]$ remains the same.

For the example of figure 1 and a transition multi-set $T_\mu = \langle \langle t_0, 2 \rangle, \langle t_1, 1 \rangle \rangle$ we can transform $M_0$ to $M_2$ as follows

$$\text{Pre}(T_\mu) = \langle \langle s_0, 2 \rangle, \langle s_1, 1 \rangle \rangle$$

$$\text{Post}(T_\mu) = \langle \langle s_2, 2 \rangle, \langle s_3, 2 \rangle \rangle$$

$$M_0 = \langle \langle s_0, 1 \rangle, \langle s_1, 2 \rangle \rangle \xrightarrow{T_\mu} \langle \langle s_2, 2 \rangle, \langle s_3, 2 \rangle \rangle = M_2.$$ 

This direct arc from $M_0$ to $M_2$ is not possible in the reachability graph of figure 2, where a sequence of steps is needed to reach $M_2$ from $M_0$.

2.1.6 Runs

The corresponding concept to the trace notion in the concurrent case is a partial order of transitions called run. A restricted place transition net and an inscription is used to express such a partial order describing the occurrence of transitions.

Definition 2.4 An occurrence net is a Petri net (see definition 2.1 page 8) with
2.1 Place Transition Nets

1. \( \forall s \in S \ | \bullet s| \leq 1 \land |s \bullet| \leq 1, \)

2. \( F \) is acyclic (\( F^* \) irreflexive) and

3. \( \forall x \in T \cup S \ \{ y \in S \cup T | (y, x) \in F \} \) is finite.

Such a restricted Petri net is also a condition event net and we use \( C \) instead of \( S, E \) instead of \( T \) and write \( K = (C, E, F) \). We define \( ^0K \) as the set \( \{ x \in C \cup E | \bullet x = \emptyset \} \) and \( K^0 \) as the set \( \{ x \in C \cup E | x \bullet = \emptyset \} \).

We annotate to each event a corresponding transition and to each condition a place using a so-called \( \Sigma \)-inscription.

**Definition 2.5** A \( \Sigma \)-inscription for a place transition net \( N = (S, T, F, w, M_0) \) and an occurrence net \( K = (C, E, F) \) is a pair of functions \( (r_C, r_E) \) with \( r_C : C \mapsto S \) and \( r_E : E \mapsto T \). For \( Q \subseteq C \) we extend \( r_C : \varnothing C \mapsto \mu S \) by \( r_C(Q) = \sum_{s \in S} |\{ c \in Q | r_C(c) = s \}| s. \)

Thus for any \( Q \subseteq C \) holds \( r_C(Q) \in \mu PLACE \) and thus it can be interpreted as a marking for \( (S, T, F, w, M_0) \).

**Definition 2.6** A run for a place transition net \( N = (S, T, F, w, M_0) \) is an occurrence net \( K = (C, E, F) \) with a corresponding \( \Sigma \)-inscription \( (r_C, r_E) \) where

\[
r_C(\emptyset K) = M_0 \quad \text{and} \quad \forall e \in E \text{ with } r_E(e) = t \text{ holds } r_C(\bullet e) = \text{Pre}(t) \text{ and } r_C(e \bullet) = \text{Post}(t).
\]

![Figure 3: A run for the place transition net of figure 1](image)

One possible finite run for the example place transition net of figure 1 is presented in figure 3. The \( \Sigma \)-inscription for the occurrence net is defined by the annotations that assign to every event \( e \in E \) a transition \( t \in T \) and to every condition \( c \in C \) a place \( s \in S \).
2.1.7 Progress and Fairness

In the theoretical treatment of Petri nets it is usual to consider only possible state changes by means of transitions and make no assumptions about their occurrence concerning fairness [Fra86]. When specifying system behavior like in our approach some kind of progress guarantee is needed. We adopt the definitions from Reisig in [Rei98] for a progress and fairness notion.

In the following we define several distinct notions of fairness and progress which rely only on the enabling and occurrence notion common for place transition nets and other net dialects. Thus the definitions of this subsection can also be applied for high level Petri nets as defined in section 2.2 or our coordination net formalism introduced in section 3. Thus we use the notion net to denote any Petri net based system in general.

The first form of progress guarantees that if a set of conflicting transitions can occur at least one will occur.

**Definition 2.7** A run \((C, E, F)\) of a net with \(\Sigma\)-inscription \((r_C, r_E)\) neglects progress for a transition \(t \in\text{TRANS}\), iff holds \(r_C(K^0) \geq \text{Pre}(t)\) (\(t\) can occur in the final situation \(K^0\)). A run respects progress for \(t\) iff it does not neglect progress of \(t\).

To have a notion also useful in a modular situation like coordination net systems we need a more powerful assumption that even when the system is alternating and an activity (transition) is not the entire time but infinitely often enabled, it has also to occur infinitely often.

**Definition 2.8** A run \((C, E, F)\) of a net with \(\Sigma\)-inscription \((r_C, r_E)\) neglects fairness for a transition \(t \in\text{TRANS}\), iff for \(t\) holds

\[
|\{ K \in [^K \Omega ] | \text{Pre}(t) \leq r_C(K) \} | = \infty \land |\{ K \in [^0 \Omega ] | \exists e \in E. \bullet e \leq K \land r_E(e) = t \} | \neq \infty.
\]

A run respects fairness for a transition \(t\) iff it does not neglect it.

A version considering a set of transitions and demands no fairness between different transitions of the set in a net can be described as follows.

**Definition 2.9** A run of a net neglects fairness for a transition set \(T \subseteq \text{TRANS}\), iff for \(T\) holds

\[
|\{ K \in [^K \Omega ] | \exists t \in T. \text{Pre}(t) \leq r_C(K) \} | = \infty \land |\{ K \in [^0 \Omega ] | \exists e \in E. \bullet e \leq K \land r_E(e) \in T \} | \neq \infty.
\]

A run respects fairness for a transition set \(T\) iff it does not neglect it.

If we not assume fairness for a system, our observations are only valid if we consider the whole net. Thus, for modular net dialects like our coordination net systems without
fairness we can not restrict our focus to a single view. However, it is essential for a modular approach, that the behavior should be predictable by considering a single view.

In order to further improve the behavior specification given by a net, we distinguish between fair transitions (the default case), which include some kind of progress guarantee as defined in any of the fairness definitions and non fair transitions which even do not guarantee any progress named quiescent, like introduced by Reisig [Rei95].

![A net with fair and quiescent transitions](image)

Figure 4: A net with fair and quiescent transitions

To specify quiescent transitions, which indicate an arbitrary nature of that transition concerning occurrence, we use grey drawn transitions as in figure 4 like Kindler [Kin97] to indicate that their occurrence if enabled is not ensured at all. So, a quiescent transition may not occur, even when it is infinite often or long enabled. The concrete semantic concerning fairness and progress for progress or quiescent transitions may vary for different net dialects.

### 2.2 High Level Petri Nets

The traditional Petri net formalism has been improved by mechanisms to extend the expressiveness and to allow to design specifications. The main directions are predicate transition nets by Genrich and Lautenbach [Gen87, GL81] which use the concept of many-sorted algebras, and colored Petri nets proposed by Jensen [Jen87] which support typing in a way programming languages usually do. We consider here in more detail the forthcoming ISO high level Petri net standard [Com97], which provides a non hierarchical high level Petri net model inspired by predicate transition nets.

Bernardinello and De Cindio represent a classification in [BC92] separating three levels of Petri nets. All nets up to normal Petri nets are classified as level one. The level 2 in this classification contains predicate transition nets and colored Petri nets. Petri nets variants providing complex tokens specified for example as abstract data types are described as level 3 nets.

We repeat here the necessary subset of the definitions from the ISO high level Petri net standard [Com97] needed to read this document on its own. For a more complete treatment we refer to the standard itself and related basic literature about predicate transition nets [Gen87] and algebraic specification techniques [EM85].

**Definition 2.10** A high level Petri net is a 7-tuple $HLPN = (S, T, C, C, \text{Pre}, \text{Post}, M_0)$ with

*...*
$S$ a finite set of elements (places)

$T$ a finite set of elements (transitions) disjoint from $S$ ($S \cap T = \emptyset$)

$C$ is a non empty finite set of types

$C : S \cup T \rightarrow C$ a function used to type places and determine a transition mode

Pre, Post : TRANS $\rightarrow \mu$PLACE are the pre- and post-mappings where

$$\text{TRANS} := \{(t, m) | t \in T, m \in C(t)\}$$

$$\text{PLACE} := \{(s, g) | s \in S, g \in C(s)\}$$

$M_0 \in \mu$PLACE is a multi-set known as the initial marking of the net.

The typing of places using $C$ corresponds to the variable typing in programming languages and assigns a literal, record or even class types to a place. In contrast the transition typing use transition modes describing the set of all possible different occurrences of that transition, e.g., if expressions are annotated to each transition the set of all possible assignments build this type.

$S$, $T$ and $C$ are restricted to finite sets and thus the structure of this Petri net notion is a finite Petri net. But by considering infinite carrier sets for sorts even when each reachable marking has a finite number of elements there can exist infinite many reachable markings.

The pre- and post-condition of a transition $t \in T$ are given by $\text{Pre}(t)$ and $\text{Post}(t)$ which can be extended to multi-sets of transitions $T_\mu \in \mu$TRANS by the linear extension

$$\text{Pre}(T_\mu) := \sum_{tr \in \text{TRANS}} T_\mu(tr) \text{Pre}(tr)$$

$$\text{Post}(T_\mu) := \sum_{tr \in \text{TRANS}} T_\mu(tr) \text{Post}(tr).$$

Enabling and occurrence rules are then given by

$$\text{Pre}(T_\mu) \leq M \quad M' = M - \text{Pre}(T_\mu) + \text{Post}(T_\mu).$$

A step notion is defined analogous to that of place transition nets.

2.2.1 From Place Transition Nets to High Level Petri Nets

The assumed simple representation of the place set PLACE as $S$ and the transition set TRANS as $T$ for the place transition nets is only possible due to the untyped nature of tokens. If places are typed at least the PLACE set has to be extended to cover this additional aspect. In the high level Petri net standard the place transition analogy is reflected by assuming a typing for places and transitions using an additional set of sorts $C$ and a typing function $C$. 

The needed simple set of sorts can be provided as $C = \{\bullet\}$ and the related typing function is simply defined by $C(x) = \{\bullet\}$ for any $x$, where $\bullet$ is simply the unique element of the unique sort $\{\bullet\}$. Thus suitable place and transition sets extended with typed values can be defined by

$$\text{PLACE}_{HLPN} = \{(s, \bullet) \mid s \in S\} \quad \text{and} \quad \text{TRANS}_{HLPN} = \{(t, \bullet) \mid t \in T\},$$

where $\bullet$ is the only element of $\{\bullet\}$. These place and transition sets are isomorph to $S$ and $T$ and thus, for example, a simple mapping $m : \mu\text{TRANS}_{HLPN} \mapsto \mu\text{TRANS}$ can be defined by

$$m(T\mu) := \sum_{(t, \bullet) \in T\mu} T\mu((t, \bullet)) t \quad \in \mu\text{TRANS}.$$

The corresponding initial marking $M_0^\bullet \in \mu\text{PLACE}_{HLPN}$ can be defined as $\forall (s, \bullet) \in \text{PLACE}_{HLPN} M_0^\bullet((s, \bullet)) := M_0(s)$.

Finally, a valid high level Petri net structure for a place transition net $(S, T, F, w, M_0)$ is given by $(S, T, C, C, \text{Pre}_{HLPN}, \text{Post}_{HLPN}, M_0)$.

### 2.2.2 High Level Petri Net Graphs

The *high level Petri net graph* notion of the standard is based on the usage of *signatures* and a *many-sorted algebra* as their interpretation. *Terms* are used as annotations for edges and transitions. In general, multi-sets of terms are used.

**Definition 2.11** A high level Petri net graph is a 5-tuple $\text{HLPNG} = (\text{NG}, \Sigma, C, \text{AN}, M_0)$ with

- $\text{NG} = (S, T, F)$ is a finite Petri net or net graph,
- $\Sigma = (R, \Omega, V)$ is a natural-boolean signature with symbols for sorts $(R)$, operations $(\Omega)$, variables $(V)$ and a corresponding $\Sigma$-algebra $H = (R_H, \Omega_H)$ where $H_r \in R_H$ denotes the carrier set\(^2\) for sort $r \in R$,
- $C : S \rightarrow R_H$ is a function used to type places,
- $\text{AN} = (A, TC)$ is a pair of net annotations where $A : F \rightarrow \text{BTERM}(\Omega \cup V)$ such that for $C(s) = H_r$ with $r \in R$ and for all $(s, t), (t', s) \in F$ $A((s, t)), A((t', s)) \in \text{BTERM}(\Omega \cup V)_r$ annotates arcs with a multi-set of terms of the same sort as the carrier associated with the arcs place and

---

\(^2\)[Com97] Note 4: places are typed by carrier sets $H_r$ for a sort $r \in R$ instead of the sort $r$ itself.
TC : T → TERM(Ω ∪ V)_{Bool} annotates transitions with Boolean terms and

\[ M_0 : S \rightarrow \bigcup_{s \in S} \mu C(s) \] such that \( \forall s \in S \ M_0(s) \in \mu C(s) \), is the initial marking function which associates a multi-set of tokens of correct type which each place.

The typing for each place is given by \( C \), but the typing or transition mode for each transition is given using an assignment \( \alpha \). Such an assignment \( \alpha \) always have to provide bindings for all free variables of the annotation terms of a transition. The sorted algebra allow then to evaluate the used terms, e.g. \( x + 1 \) or \( x/y \). By type the transitions with modes describing the possible assignments for the free variables of the annotations of the transition itself and all connected edges the semantic for high level Petri net graphs is provided. An assignment is a set \( \{ \alpha_r | r \in R \} \) of assignments \( \alpha_r \) for variables of sort \( r \in R \) with

\[ \alpha_r : V^r \rightarrow H_r. \]

The extension \( [tr]_{r}^{\alpha_r} \) of a single term \( tr \) of sort \( r \in R \) is given by

\[ [ ]_{r}^{\alpha_r} : \text{TERM}(\Omega \cup V) \rightarrow H_r \]

and an extension to multi-sets over terms \( T_{\mu} \in \text{BTERM}(\Omega \cup V) \) of any sort \( r \in R \) is given by \( \text{Val}_\alpha : \text{BTERM}(\Omega \cup V) \rightarrow \mu H \) defined by

\[ \text{Val}_\alpha(T_{\mu}) = \sum_{tr \in T_{\mu} \land tr \in \text{TERM}(\Omega \cup V)_{r}} T_{\mu}(tr) \ [tr]_{r}^{\alpha_r}. \]

But not all assignments \( \alpha \) result in a correct situation and thus we restrict them to valid cases by demanding that

\[ \forall tr \in TC(t) \ [tr]_{r}^{\alpha_r} = \text{true}. \]

We define corresponding to the high level Petri nets the pre- and post-conditions using the multi-set cross product extension \( \times \) as follows

\[ \text{Pre}(t) := \sum_{(s,t) \in F} s \times \text{Val}_\alpha(A(s,t)) \quad \text{Post}(t) := \sum_{(t,s) \in F} s \times \text{Val}_\alpha(A(t,s)). \]

Thus the enabling and occurrence rules for a given valid assignment (mode) are conform derived from the high level Petri net semantic using the Pre and Post mappings. The definitions for a single transition \( t \in \text{TRANS} \) can be extended to multi-sets of transitions \( \mu \text{TRANS} \) in order to cover also concurrent steps.

See the example presented in figure 5 of a high level Petri net graph, where an integer variable \( x \) is used. The places \( s_1, s_2 \) and \( s_3 \) are all of type \( \text{int} \) and transition \( t_1 \) has as pre-condition any integer value from place \( s_1 \). When firing, this value is bound to \( x \) and thus the resulting value for the expressions \( x + 1 \) and \( x - 1 \) are also determined using a corresponding assignment \( \alpha \) with \( \alpha(x) \in \text{int} \). For example the marking \( \langle \langle s_1, 3 \rangle, 1 \rangle \)
2.3 Extensions for Modularity

Historically often the missing modularity is criticized. So several extension have been studied which try to solve this problem. Colored Petri Nets are extended by Jensen in [Jen90] to hierarchical colored Petri nets which support a modular description. Huber, Jensen and Shapiro demonstrates in [HJS90] that a number of different strategies can be used to build a modular system. The possible mechanisms are substitution of transitions or places, invocation transitions and fusion sets for transitions or places. All these mechanisms, excluding the invocation, are very Petri net specific and do not rely on the natural structure of the net.

\[
\begin{align*}
\text{int} & \quad x \\
& \quad t_1 \\
\text{int} & \quad x + 1 \\
\text{int} & \quad s_2 \\
\text{int} & \quad s_1 \\
\text{int} & \quad x - 1 \\
\text{int} & \quad s_3
\end{align*}
\]

Figure 5: A high level Petri net example

\[
\begin{align*}
\text{int} & \quad y|x \wedge y > 1 \\
\text{int} & \quad y \\
\text{int} & \quad x/y \\
\text{int} & \quad x \\
\text{int} & \quad s_1 \\
\text{int} & \quad s_3
\end{align*}
\]

Figure 6: A second high level Petri net example

may be transformed by the occurrence of \( t_1 \) with assignment \( \alpha(x) = 3 \) to the marking \( \langle ((s_2, 4), 1), ((s_3, 2), 1) \rangle \).

As demonstrated in the example net of figure 6, the concept is even more general. The assignment provides a binding for each variable used in annotations to the pre- and post-condition edges or annotations to the transition itself. All possible assignments are assumed to be the mode (typing) for that transition. Several pre-condition edge annotations may use the same variable or even an arbitrary one only restricted using a guard expression can be used. Thus, for example, the non deterministic choice of a suitable value can be used to specify several behaviors in an abstract and powerful way. In the example of figure 6, any arbitrary chosen integer value for integer variable \( y \) of a valid assignment must fulfill the guard condition \( y|x \wedge y > 1 \) and thus \( y \) is a divisor for \( x \). The described net starts with \( \langle ((s_1, n), 1) \rangle \) and processes finally all prime factors with correct multiplicity for \( n \) by \( \langle ((s_1, p_1), \pi_1(n)), \ldots ((s_1, p_i), \pi_i(n)), \ldots \rangle \), a known difficult computational problem. Note that even not decidable problems can be expressed using such guard expressions depending on the given \( \Sigma \)-algebra and its operations.

3\text{we do not use the notation assign}_{\text{r}} \text{ of} [\text{Com97}] \text{ to allow to distinguish different assignments} \alpha

4\text{we denote the number of occurrences of} tr \text{ in} T_\mu \text{ directly using} T_\mu(tr) \text{ instead of} \text{mult}(tr, T_\mu)
notion of information exchange directly, but encode it into a net specific view. Only transition invocation can be considered as the procedural abstraction common in programming languages.

2.3.1 Object-Oriented Nets

Based on different extended Petri nets classes, approaches to integrate the ideas of object-orientation into a Petri net arises. A separated object model represented by tokens is integrated into nets in the ExSpect approach by Van Hee, Rambags and Verkoulen [vHRV91], where two separate models are build and the nets control algebraically specified objects. The PROT approach [BB88] allows to model a system by a hierarchy of subnets connected by places like in high level Petri nets. Places are FIFO like queues and the elements of the nets are refined using a programming language. We classify PROT as an object based system. The elementary object nets presented by Valk [Val98b, Val98a] also support only two levels and thus do not support dynamic issues and inheritance. These approaches can be considered to be object based.

Object orientation itself is introduced in several ways. Some try to extend and combine algebraic specification with nets. Battiston et al. [BCM88] propose OBJSA nets which integrate an object-oriented data model with the synchronization of superposed automata nets. Buchs et al. [BB94, Gue95, BB97] extend algebraic data types to COOPNs, by associate methods with transitions and synchronizing them. These approaches are abstract data type dominated and use non-hierarchical combination strategies like transition fusion based on Petri nets to describe the concurrency aspects. We think the concurrency modeling has to be integrated more directly into the object structures in order to allow the refinement of subsystems. It should support a way to specify the behavior for different tasks in separation.

Lakos [Lak94, Lak95] extends high level Petri nets to OP-Nets (predecessor LOOPN, LOOPN++) by identifying sub pages as classes and defining inheritance as some kind of net refinement. Communicating nets by Sibertin Blanc [SB94] represent objects by nets which communicate via places. Cooperating nets are based on this modular message passing oriented nets and offer a client/server like cooperation mechanism using a call and return transition as introduced in [SB93]. Moldt et al. demonstrated in [MM97, DIM97] with OCP nets, that traditional modeling notations for object-oriented analysis and design can be replaced in principle by nets. However, OCP nets use interaction principles (message passing) which are too low-level, resulting in missing abstraction. These approaches come up with very technical and Petri net specific methods for object interaction, which are all covered by the suggested combination methods for high level Petri nets. The object behavior is specified using a single net and thus different tasks of an object can not be specified in separation.

A smalltalk specific approach integrating nets into the object-oriented smalltalk language is PNtalk [CJV97] build for rapid prototyping. Object tasks are separated and calls are
the basic way to interact. The untyped nature is in contrast to our idea of an extended typing notion to improve separation.

2.4 Extensions for Interaction

Besides these ”main stream” extensions to Petri nets there are several other directions that extend the traditional Petri net approach concerning its semantic and interaction.

2.4.1 Asynchronous and Synchronous Communication/FIFO

Common technologies for distributed object systems like CORBA, DCOM or the Java language extension RMI provide only the asynchronous operation call with or without asynchronous reply as basic interaction scheme. Thus asynchronous communication is one essential aspect our formalism should cover. But synchronous interaction is also useful, because it provides a higher level of interaction needed, for example, to combine preconditions and operation execution on the semantic level. A synchronous interaction has to be emulated when a distributed object bus as implementation platform is wanted. Thus servers capable of interacting in a synchronous fashion can be build. But to achieve full compatibility also restricted servers that do not support synchronous interaction are useful. Even special performance or reliability requirements might enforce to restrict the interaction to the asynchronous case. Consider, for example, a system wide heavily used server which should guarantee sufficient throughput and minimal response times. For such a system, synchronous interaction which may result in multi-party synchronizations and several forms of delays may be not acceptable.

The needed asynchronous communication must ensure some order preserving and thus FIFO Petri nets [MF85, Rou87] may be considered, too. But in general, we want to avoid adding FIFO places as a general concept for the nets and only use them to model the communication subsystem layer. It is necessary to abstract from this layer as much as possible and, hence, the work of Souissi and Memmi [SM90] where FIFO Petri nets are used as a medium should be considered for this purpose.

2.4.2 Procedure Calls

The procedure call or operation call in object-oriented systems is the standard mechanism of interaction in programming languages. Hence, our approach has to support it anyway. There is some basic theoretical work about Petri nets providing procedures by Kiehn [Kie90], but this work does not cover object interaction.

For the PEP tool environment [BG96, Gra97] of Best et al. the extension of B(PN)$^2$ [BH93] with procedures and a mapping to the underlying M-Nets [BFH+98] formalism has been presented by Lilius and Pelz [LP96] and Fleischhack and Grahlmann in [FG97].
2.4.3 Port Passing Calculi

The development of the port passing calculi started with Nielsen’s and Engberg’s work [NE86] and was reformulated by Milner and his colleagues in [MPW89] to the well know \( \pi \)-Calculus. He further improved it in [Mil90]. For an overview see the tutorial [Mil93]. A visualization approach for it is presented by Milner [Mil94]. For the background work on asynchronous port passing approaches see the actor model from Hewitt et al. [HBS73, Hew77, Agh86]. The \( \pi \)-calculus has often been used when dynamic port passing communicating systems are described and it is capable of express the dynamic of object-oriented approaches. Honda and Tokoro [HT91] build a first asynchronous object calculus based on the ideas of the \( \pi \)-calculus. Other approaches are Vasconcelos [Vas94], Puntigam [Pun97] or Najm et al. [NNS99].

In relation with Petri nets there has been done some work to establish a multi-set or Petri net semantic for the \( \pi \)-calculus by Engelfriet [Eng94, Eng96] and Busi and Gorrieri [BG95]. A net based calculus named Mobile Nets by Busi is presented in [Bus99, AB96]. They extend the \( \pi \)-calculus to contain true concurrency in a net based fashion, but their focus is not to extend Petri nets including the graphical representation to cover dynamic port passing systems. Instead they extend the usual textual binding for \( \pi \)-calculus processes to cover some notion for places that can be accessed in parallel.

2.5 Design Decisions

We think it is more promising for the system design with Petri nets to avoid a Petri net specific mechanism and integrate Petri nets into an usual object-oriented decomposed system. The approach should rely on the well studied and successful mechanisms for abstraction and encapsulation. No current Petri net approach provides a suitable visual notation for our purpose and their intention is often of more theoretical concern. Another reason for choosing none of the above current object-oriented Petri net notions is their lack of flexibility. Often they introduce a Petri net oriented structural view and are not able to deal with multiple interfaces, port passing or even dynamic net creation.

We decided to start not directly with an object-oriented net notion and instead build a flexible net notion (coordination nets) providing the necessary features needed for our object-oriented net dialect specified in section 5.

From an engineering point of view it is useful to rely on existing standards, so we decided to base our formalism on the forthcoming ISO high level Petri net standard [Com97], which provides a non hierarchical high level Petri net model inspired by predicate transition nets [Gen87]. We extend the non hierarchical high level Petri net notion based on the basic mechanism provided by the \( \pi \)-calculus.
3 Coordination Nets

3.1 Basic Concept

In extension to high level Petri nets as defined in the standard we need some kind of modularity as an essential addition for our basic formalism. To achieve this, we build a system based on a set of coordination net graphs and allow them to interact.

3.1.1 Requirements

![Figure 7: Requirements for modeling an object](image)

The formalism should allow to model multiple instances of one object type, each one providing a set of interfaces with dynamically changing external protocol state. The expected situation for an object is visualized in figure 7. Each object must handle possibly multiple exported interfaces and can use the imported ones to fulfill the requests. Each object instance may process several tasks in a more or less independent fashion. Assuming a single net specifying the whole object behavior is not useful, instead there should be a mechanism to describe separate aspects in isolation. For each connection the processing should provide some guarantees concerning the request processing order to avoid unexpected behavior by reordered requests. Thus some kind of FIFO behavior for the message buffers is needed.

We allow several net instances to realize the object behavior together and thus a mechanism like place sharing for them is necessary to model the object environment shared by all net instances of an object instance. The concept of port passing and dynamic net creation of the π-Calculus can then be used as background model. We do not provide a build in object notion in our basic net dialect coordination nets. Instead, we provide an interface based separation using typed ports which consists of an interface and a protocol restricting the message occurrences. Our second net dialect object coordination nets will provide a suitable object and class notion based on the first one in section 5.

3.1.2 Architecture

We have to extend the standard by suitable mechanisms for communication, place sharing as well as dynamic net and port creation. In figure 8 the basic structure of a coordination net system is shown. There do exist multiple instances of the same net type possibly sharing places between a net instance and its child ("real place arrow connection). The
nets may communicate using a communication infrastructure. We visualize it using a bus like architecture as, for example, in CORBA or DCOM. An object may be realized using a dynamically changing set of net instances (see dashed box in figure 8), where the associated objects and exported interfaces are represented by special port token.

### 3.1.3 Communication

The basic idea for communication is to introduce ports $\zeta$, $\eta$, … representing associated or exported interfaces (objects) as pairs of connected communication endpoints. These ports can be used to receive a message using the following annotation for a transition

$$\eta = \zeta?\langle\text{op}(a_1, \ldots, a_n)\rangle,$$

where $\eta$ is the resulting port and $\langle\rangle$ denotes a given marshaling function. $\text{op}(a_1, \ldots, a_n)$ stands for an operation call with operation name $\text{op}$ and input parameters $a_1, \ldots, a_n$. There may be several distinct return vectors for a call and thus we use $\overline{\text{op}}_i$ as operation name for the return alternative $i$ to an operation $\text{op}$ and annotate $\overline{\text{op}}_i(r_1, \ldots, r_m)$ for a reply with return vector $r_1, \ldots, r_m$.

A corresponding synchronous send can be specified using a port $\zeta$ and the synchronous send operator $\downarrow_s$. Analogous an asynchronous send can be specified using $\downarrow_a$,

$$\eta = \zeta\downarrow_s\langle\text{op}(\ldots)\rangle \quad \eta = \zeta\downarrow_a\langle\text{op}(\ldots)\rangle.$$
3.1 Basic Concept

We distinguish *provide ports* $\rho$, $\varphi$, ... for exported and *usage ports* $\phi$, $\varphi$, ... for associated interfaces. A provide port can receive operation calls $\text{op}$ and sends replies $\overline{\text{op}}_i$ whereas an usage port can be used to send requests $\text{op}$ and receive replies $\overline{\text{op}}$. We distinguish asynchronous and synchronous interaction, because the synchronous interaction provides more sophisticated ways to interact. The asynchronous communication is in contrast more efficient and reduces the coupling between two systems and thus a provided port may restrict sending to the asynchronous case when a server does not provide synchronous interaction.

$$\eta = \zeta(\text{!}(\text{op}(...)))$$

Later we will omit such a detailed handling when protocol nets for ports are considered. Synchronous or asynchronous interaction is only considered as a background model to describe the semantic of the more general send operation (!).

### 3.1.4 Create

$$\phi = @N \quad (\rho, \phi) = @P$$

To create ports of type $P$ or net instances for a declared net type $N$ also corresponding annotation expressions are supported. A created net expression $(\phi = @N)$ binds to $\phi$ an usage port corresponding to a special initial provided port $(\rho_{\text{std}}; \text{std})$ each new net instance contains. This initial connection allows to establish more connections by using these port connections to publish other ones. After a port creation $(\rho, \phi) = @P$ a pair of new unique connected usage and provide port instances is bound to $\phi$ and $\rho$.

### 3.1.5 Example

![Figure 9: A coordination net graph example](image-url)
By allowing the described annotations in net declarations, a dynamically changing set of net instances interacting via port instances can be specified. To achieve a better visual representation we draw all transitions annotated with receive terms and imported places with a shadow as shown in figure 9. There is a request received in transition 1 which is replied with a send expression in transition 2. Transition 3 creates a new net of type $N$ and propagates the resulting usage port $\phi$ together with its other pre-conditions $a_1$ and $a_2$ in form of a vector to a place. Transition 4 may consume it then. Hence, the parts which a single coordination net graph distinguishes from a high level Petri net as defined in the high level Petri net standard are the additional annotations. They add message send, message receive, port creation and net creation to a transition as shown in figure 9.

The single net graphs are interacting via send and receive annotations which are using the already introduced ports as addresses. The resulting system consist of a number of net graph instances connected via ports and a marking for all of them, as presented in figure 10. The left two nets interact via corresponding usage/provide ports and a synchronized send and receive transition pair. The synchronous send ensures, that the message is directly received. In the middle a net creation is presented and the resulting port pair is visualized. The imported place of the created Net $N$ is linked to the corresponding local place of the creating net. A port creation is demonstrated in the right net.

An asynchronous send is visualized in figure 11. Here the send message is first appended to the message queue and later received. Such a message queue exist for every usage port.
3.2 Typing Concept

A suitable typing has to carefully distinguish usual types (literals) that describe a value domain and represent passive data and ports which are handles or identifiers that allow to request certain activities or attributes.

3.2.1 Literal Typing

The typing for literals can be handled using the basic algebraic formalisms using sorts and functions. For records, a subtype notion can be build based on the set of attributes or fields, but when also the exchange of records between different entities is concerned, the additional attributes and their storing may result in several problems and thus we omit this possibility in conformance with CORBA or DCOM.

3.2.2 Port Typing

For ports a typing that supports subtype polymorphism is essential. Ports represent connections to other entities in a fashion that should ensure abstraction and autonomy which are the essential characteristics of object based systems (see Wegner [Weg87]). By additionally providing a notion of inheritance, the typing can be considered to fulfill the requirements for an object-oriented language.

The basic idea for port typing is to associate an interface (signature) and a behavior to every port connection. Thus the object life cycle and the possible interaction with an

Figure 11: An example for an asynchronous communication
object become a part of the usage contract. In this version of the report we will restrict the focus on the general typing scheme and omit details concerning the inheritance notion and subtyping.

**Interfaces**

![Diagram of an interface](image)

interface I {
    m(int a1, int a2);
}

Figure 12: Syntactically restricted message formats on a port

By restricting the messages that can be sent to a port of specific type, the occurrence of *message not understood* errors can be excluded. An *interface* belonging to every port is used to guarantee that every *user* or *client* knows which messages the *provider* or *server* will understand. In traditional sequential programming languages and interface description languages (see CORBA [Sie96, Obj95, Obj96], DCOM [Cha96] and DCE [Cha94]) these restrictions are based on syntactical declarations of allowed message formats. They are usually introduce operation names or comparable message tags like those shown in the example in figure 12.

**Behavior and Contracts**

Additionally, the processing of a *server* may depend on its internal state. Such dependencies can lead to *never processed messages*, when the server never reaches a state where it can process a certain request or other non processed messages may block the server forever. Depending on the implemented scheme of message processing a server may detect that a message in its queue can not be processed and refuse the processing by replying a *can not process message* error or the server will not even detect that its message processing results in a stall. Thus several effects of concurrent systems like *deadlock*, *starvation* or *live lock* or incompatible connections between clients and servers may result in system malfunctions that should be excluded.

By adding behavioral information to the type system not all these problems can be tackled. There are several more or less local aspects like message formats and the general supported sequences of interactions that may improve the handling while other aspects like the overall synchronization have a too global focus to be attached to port types.

The supported combination of a *syntactical interface* and a description of guaranteed be-
3.2 Typing Concept

behavior is named *contract*. We consider several relevant examples for interaction protocols of *contracts* in the following to describe the intended behavior aspect for our port typing.

**Remote Procedure Call**

![Figure 13: The client server connection for a remote procedure call](image)

One basic kind of interaction is the *remote procedure call*. It provides the basis for *client/server* based systems. The structural situation is visualized in figure 13. The assertions contained in an usual *remote procedure call* contract are:

- for every request exactly one reply is send and
- the reply is provided after a finite amount of time.

A protocol describing these assertions can be formulated using a labeled place transition net with *progress* and *quiescent* transitions restricted to a state machine net. It specifies which states between a *caller* and a *callee* are processed during the call. The states of the *remote procedure call* protocol presented in figure 14 are given by the markings (starting with $s_0$ marked) and the interaction is given by the resulting *failure trace* model. Both sides of such a *remote procedure call* protocol have somehow redundant views of the protocol. Each *optional behavior* for the *caller* (*quiescent transition*) corresponds to an *obligation* for the *callee* (*progress transition*) and every *obligation* for the *caller* to receive an answer is also an *obligation* for the *callee* to provide the reply. Thus, we will further only consider the view of the *callee*. The provider of the protocol has to guarantee that in state $s_0$ a request $m$ will be accepted and that in response a reply $m'$ will be produced. The usage side can arbitrarily send requests $m$ as long as the protocol state allows this. Afterwards in this *remote procedure call* case the usage side has to guarantee that every possible reply will be received. The state $s_1$ is *temporary* in the sense that the providing side has to send $m'$ after a finite amount of time specified using a *progress* transition and the using side has to accept this reply and thus $s_1$ represents no *durable* situation. In contrast, $s_0$ can only be left using a *quiescent* transition and thus the state is *durable*. A state with *quiescent* and *progress* transitions is either *durable* nor *temporary*. This kind of intermixed status is named *transient*.

A protocol conform usage is presented in figure 15 where the port $\phi$ is transformed to $\varphi$ by sending $m$ and later transformed to $\chi$ when receiving the reply $m'$. The port type and state is annotated using the shortcut $I[s_i]$, where $I$ denotes the *interface* and *protocol* and $s_i$ the specific protocol state.
The example protocol of figure 14 supports only a single call. To obtain a protocol that provides a service \( m \) infinite often we can simple unite \( s_2 \) and \( s_0 \). A protocol providing several distinct calls may be build by simply sharing \( s_0 \) for all these single distinct copies of the simple cyclic remote procedure call protocol. Such a general scheme for a protocol providing services \texttt{open}, \texttt{read} and \texttt{close} is presented in figure 16. But the protocol is still semantically \textit{stateless}, because the states \( s_1 \), \( s_2 \) and \( s_3 \) do only represent temporary states which express that a request of concerning type has to be replied before further request should be processed.

In practice, for example, a file handle protocol will imply a certain usage, e.g. no read request will succeed when not an open request has succeeded before or when the end of file is already reached. A more appropriate protocol for a file handle with operations \texttt{open},
3.2 Typing Concept

![Diagram](image)

**Figure 16**: General protocol for multiple supported *remote procedure calls*

![Diagram](image)

**Figure 17**: A realistic protocol for a file handle

*read* and *close* is presented in figure 17. The initial state where only an *open* request is possible is named *[closed]*. One possible reply is *open*₁ as an acknowledgment for a successful opening which results in state *[open]*. If the request fails, an *exception* *open*ₑ is replied and the protocol remains in state *[closed]*. A file handle in state *[open]* can further be used to read data. A successful *read* request is signaled by reply *read*₁ whereas the reached end of file results in *read*₂ and a state change to *[EOF]*. If we are either in state *[open]* or *[EOF]* the *close* request can be used to close the file handle and reach the state *[closed]* again. An even more sophisticated treatment is often useful, when no
synchronization between different replies for several calls is wanted.

### 3.2.3 Remote Procedure Call with Asynchronous Reply

![Diagram of caller and callee connections](image)

**Figure 18:** Connection situation during a running request

To describe a remote procedure call where the reply occurs in an asynchronous fashion we have to use a second new generated port to represent the resulting situation that a second independent connection is established temporary. In contrast to the usual remote procedure call, this time the contract between the caller and callee is split up. See figure 18 for an overview of the new structural situation.

![Protocols of remote procedure call](image)

**Figure 19:** Protocols of a remote procedure call with asynchronous reply

In figure 19 the protocol for $I$ and $I'$ are specified. Both are the version from the callee perspective where $I'$ is also embedded into $I$ protocol. On the right side the contract for the temporary port with interface $I'$ specifies that the server is restricted to send the return only once and ensures, that the return message will be received by the caller. Thus a non terminating call or a pending answer, that will never be received and may block the callee, is excluded.

The usage side for a remote procedure call with asynchronous reply is presented in figure 20, where additionally the implicitly defined interface $I'$ is shown. The life cycle of such a port of type $I'$ is limited to one message and thus we call it a throwaway port. The throwaway interfaces are usually omitted, because their construction is obvious and they contain no useful additional information (see figure 20 right-hand-side).
3.2 Typing Concept

3.2.4 Sharing a Provided Port

Stateless protocols like presented in figure 16 can be shared without any coordination when each usage client guarantees that every interaction is continued until a durable state is reached. Then each client can simply use the protocol without considering any coordination with other clients. In contrast, in the file example of figure 17, sharing can not be simply allowed. We have to coordinate the usage, because multiple concurrent usages may lead to not protocol conform usage. In special cases it can even be semantically critical to share a protocol. In the file example, we may decide that arbitrary interleaved access to a sequential file device may result in non deterministic results and thus has to be avoided. Thus, we distinguish shared and exclusive ports at creation time by different types. For the usage side arbitrary interference with the other usage ports is assumed for the shared case.

3.2.5 Actions and Abstraction

To simplify the specification of protocols we use macros to describe these typical steps of behavioral protocols in a more compact visual fashion. We use the usage side perspective for these macro definitions to support a visual embedding of contracts when they are used. We use so called activation edges drawn with a white head to describe calls in the protocol. The edges for internal steps are also using the white heads to achieve a visual simple appearance and simply denote the protocol flow using this kind of edges.

The macro for a remote procedure call is presented in figure 21, where the call and reply steps are combined to a single symbol. During such a remote procedure call the protocol state is not accessible for other transitions then the return transition. When more then one alternative return transition exists we split up the return part of the action symbol as

```java
interface I { 
  m(int a1, int a2, I');
}

interface I' { 
  m(int r1);
}

abbreviation:

interface I {
  m(int a1, int a2) 
  -> (int r1);
}
```

Figure 20: A call and return with temporary return port
Figure 21: A macro for a call abstraction presented in figure 21 right-hand-side.

Figure 22: A macro for a call with asynchronous reply

In the case of a remote procedure call with asynchronous reply nearly the same symbol can be used as presented in figure 22. Only the start of the leaving edge at the first or second part of the symbols reflects the distinction. The usual remote procedure call thus block all other interaction until a reply arrives while the remote procedure call with asynchronous reply does not. Note that the macro from figure 22 extend the needed net class for protocols beyond finite state machine nets.

Figure 23: A macro for an one way call

When no reply is needed the remote procedure call with synchronous or asynchronous reply introduces some kind of overhead. To avoid this, CORBA supports so called one way calls. For this case of an one way call we also provide a macro defined in figure 23.

Figure 24: Macros for different internal steps

The internal steps of the providing side can be expressed using unlabeled quiescent or progress transitions. Macros which use a shadow to indicate that their occurrence is controlled by the providing side (externally) are defined in figure 24.
3.2 Typing Concept

To represent durable or transient states we provide a resource macro (see figure 25). It is used to achieve a higher degree of conformance with prototypical net macros that specify such basic steps for the usage or providing side. By using nearly the same graphical representation in the protocol and usage case and a slightly different one for the providing side a seamless integration for protocols can be achieved. In section 4 we define the needed behavioral abstractions for protocol usage and providing and, for example, macros for a general remote procedure call on the usage side and provide side are presented.

3.2.6 Conformance

In general unlabeled steps are used to abstract from shared usage or autonomous object activity. They lead to several consistency problems concerning the contract state. E.g., the usage side(s) can not know the exact state of the connected provide port. For these cases, the provided synchronization is of great importance to achieve a sufficient interaction. We will further ignore any asynchronous emulation schemes for synchronous interaction and instead simply restrict our considerations to the availability of operation calls and abstract from the needed synchronization mechanism. By abstracting from the technical details like the need for asynchronous or synchronous interaction, the notation is more suitable for the analysis and design levels.

See the example in figure 26 where a protocol with an internal step is described. In the initial state [init] the operation m is available. Once the operation m has taken place, the only knowledge the usage side can have is that the protocol is either in state [tmp] or [final].
For \text{tmp} the operation \text{n} can not be accessed secure in an asynchronous fashion, because always the \( \epsilon \) step to \text{final} is possible. Also the availability of operation \text{o} in state \text{final} can only be detected in a synchronous way due to the unknown current state after operation \text{m} has occurred.

To overcome such error prone and explicit handling of synchronous and asynchronous interaction we abstract from these details. Hence, the usage side is modeled using the set of possible provide port states. Asynchronous interaction can only take place when every not transient state of the state set guarantees the availability of the specified operation. In contrast, when \text{synchronization} with the providing side is assumed, any interaction available for \text{at least one} element can be tried and will succeed if possible. Synchronizing this way, we can interact in a secure fashion even when the correct state is not known. By specifying a set of possible synchronous interactions we can even achieve a secure coordination for each possible providing side behavior which is protocol conform.

![Coordination Net Diagram](image)

Figure 27: The file protocol specified using the macros

The protocol of a file handle from figure 17 specified using the protocol macros in figure 27 demonstrated the obvious benefits. The structure and remote procedure calls are more obvious.

### 3.3 Coordination Nets

We have to extend the basic sort or type sets of the high level Petri net standard to fulfill the additional requirements for our coordination net system. A mapping of such a coordination net system to a standard high level Petri net is provided in appendix section A. The level 2 conformance of our approach with the standard is demonstrated there.
3.3 Coordination Nets

3.3.1 Types

We assume a basic type set \( R \) and a set of operations \( \Omega \) containing a generalized basic type system of a middleware approach like CORBA or DCOM including at least a type \( \text{Bool} \). We will further extend it to specify communication ports and message contents.

Unique Identifiers

Adding dynamic creation of ports and nets to the relative static high level Petri net formalism is an essential difficulty for the whole formalism. We can use some kind of counter to generate unique numbers to generate also unique net indices and port numbers, but a single counter limits the parallel occurrence and thus we instead assume an infinite set of elements \( r \) of a sort \( \text{REVERSE} \). We will encode it using a multi-set \( \mu \text{REVERSE} \), but restrict it to be a set. Thus the assignments can be used to choose a suitable value from the reverse and two concurrent occurrences of the same chosen reverse element can be excluded. See the evolving algebra approach [Gur95] for the theoretic background for this reverse universe construction. For an efficient mapping for a simulator or analysis tool the dialect should use a construction using counters or even try to exploit the resulting symmetries.

Net Identifier

To identify different instances of nets we introduce a special set of sorts \( \text{Id} \) with an element \( \text{Id}_N \) for each net type \( N \) of the net type set \( N \). A function \( \text{idGen}: \text{Id} \times N \times \text{REVERSE} \rightarrow \text{Id} \) with \( \text{idGen}(id, N, r) \in \text{Id}_N \) and for any \( r \neq r' \) holds \( \text{idGen}(id, N, r) \neq \text{idGen}(id', N', r') \) is used to generate new unique net indices based on a given unique element of the \( \text{REVERSE} \). A given net index \( id \in \text{Id} \) additionally provides access to the parent id (\( \text{idParent}(\text{idGen}(id, N, r)) \)), the net type (\( \text{idType}(\text{idGen}(id, N, r)) = N \)) and the used unique number (\( \text{idReverse}(\text{idGen}(id, N, r)) = r \)).

Ports

Additionally we assume a set of sorts for provide ports \( \text{P}_p \) and usage ports \( \text{P}_u \) building together the set of all port sorts (types) \( \text{P} \). An interface and all reachable states of the protocol together build a protocol state set \( \text{P} \subseteq \text{P} \). We may distinguish the different types for each specific state using restrictions like \( \mathcal{P}[s] \) to ensure that the described port type is in protocol state \( s \). We use \( s(P) \) to denote the corresponding state. Values (constants) of type \( P \in \text{P}_p \) are written \( p^x_y \) and \( u^x_U \) for \( U \in \text{P}_u \). We use the Greek letters \( \rho_p, \varphi_p, \sigma_p, \zeta_p, \ldots \) as variables for a provide port and \( \psi_U, \varphi_U, \chi_U, \psi_U, \ldots \) for an usage port. Often the type comment is omitted when it is obvious from the context. If it is not clear that a provide or usage port is given, we use the variables \( \zeta, \eta, \theta, \vartheta \). The alphabet \( \mathcal{A}_P \) of a port state set is determined by all operations declared in the interface, but not all of them may be accessible in every protocol state. Every usage of a port may change the ports
protocol state and thus every time a port is used by sending or receiving a message, only the resulting new port state represents the connection correctly.

Shared and Exclusive Ports

To further distinguish exclusive and shared usage ports, we identify usage port values $p_r^g$ as exclusive iff $r = q$ and shared otherwise. Providing ports or exclusive usage ports cannot be copied. Shared usage ports cannot be copied by cloning them. The resulting new port and the old one are both accessible afterwards.

Create Ports

When an exclusive port pair is created, the corresponding provide and usage port values are $p_r^r$ and $u_{r'}^r$ for a new unique $r \in \text{REVERSE}$, a provide port type $P$ and usage port type $U = \bar{P}$. For a net creation with $id = idGen(id', N, r)$ with an exclusive port type $\text{netType}(N)$ the resulting usage port is $u_{r'}^r$ and the standard providing port of the new net is $p_r^r$. In the shared case a second element $q$ of the $\text{REVERSE}$ is needed and we always create a pair $p_r^r$ and $u_{r'}^q$.

Message Encoding

A sort $\text{Msg}$ for the bijective encoding of operation with a parameter vector (tagged messages) using a marshaling function $\langle \rangle : \text{TERM}(op) \rightarrow \text{Msg}$ is assumed. For an operation $m$ and $\bar{a}$ a valid argument vector the message encoding evaluates $\langle m(\bar{a}) \rangle$ to an unique value of $\text{Msg}$. Note that no variables for operation names are allowed.

Especially for the communication, the set of sorts $R^c$ correspond to $R_H^c = R_H^p \cup \{\text{Msg}\} \cup P \cup P_u \subset R_H$ as basic type system for the communication system is needed. We do not extend $R_H^c$ with $\text{Msg}^*$, $\text{Id}$ or $\text{REVERSE}$, because these sorts and carrier sets should only occur in the technical transition annotations and not in edge annotations or as message parameters. For every sort $R$ we specify its technical extension sort set $R^+$ corresponding to $R_H^+ := R_H \cup \{\text{Msg}^*, \text{Id}, \text{REVERSE}\}$. Corresponding operator $\Omega^+$ and variable sets $V^+$ are assumed.

Asynchronous Communication

The asynchronous communication mechanism is based on FIFO queues. These are build using a set QUEUE of pairs $(\phi, w)$ assigning a corresponding message word $w \in \text{Msg}^*$ to an usage port $\phi$. Thus we have the following sort for queues

$$\text{QUEUE} = \bigcup_{U \in P_u} U \times \text{Msg}^*.$$
3.3 Coordination Nets

The system message queues are then build by the corresponding multi-set $\mu$\textit{QUEUE}, which is restricted to be a set $(\forall \phi \in \bigcup_{U \in \mathcal{U}} U.B(\phi) \leq 1)$ to ensure that for every usage port at most one message queue exists. Even for any two ports with the same identity holds this restriction which can formally be described by $(\forall U, x, y \; \sum_{U \in \mathcal{U}} B(u_{U}^{x,y}) \leq 1)$. The multi-set encoding is necessary to be able to also express pre-conditions with conflicting port queue accesses $(B(\phi) \geq 2)$, but true concurrent access on a queue is not possible, due to the inherent sequential nature of queue behavior.

3.3.2 Port Typing

In addition to the interface part, a port type corresponds simply to a state of its protocol and thus we use $P \xrightarrow{m} P'$ for two provide port types $P, P' \in \mathcal{P}$ to denote that for the corresponding states such a step is possible. A protocol and interface are initially represented by a provide port type that corresponds to the initial state $M_{0}$ of the protocol and all reachable provide port states are corresponding to a state $M \in [M_{0}]$. $s(P)$ denotes the corresponding marking (state).

Provide Ports

The view of the usage side is slightly more restricted and the alphabet $\mathcal{A}_{\text{P}}$ might be partitioned into visible operations $\mathcal{A}_{\text{P}}^{\text{v}}$ and a set of internal steps $\{\epsilon_{1}, \epsilon_{2}, \ldots\}$. Thus the considered labeling for the usage side is simply adjusted to map all internal steps $\epsilon_{i}$ to $\epsilon$ and thus the usage side can not distinguish them. The providing side instead needs a way to distinguish them and thus the different internal steps are visible. By convention we use $\epsilon_{P,P'}$ to denote the $\epsilon_{i}$ which transforms the protocol from state $s(P)$ to $s(P')$.

Exclusive Usage Ports

For the exclusive usage side, we have to extend the step marking to cover sets of markings. Thus an usage port type corresponds to a set of states and the initial marking is that one corresponding to all initially reachable states $\{M' \in [M_{0}] | M_{0} \xrightarrow{\epsilon} s.M'\}$. If for an exclusive standard provide port of a newly created net with type $P$ the corresponding usage port $U = P$ has to be determined, we use simply the state set $s(U) = \{M' \in [M] | M \xrightarrow{\epsilon} s.M'\}$ for $M = s(P)$.

Definition 3.1 For two exclusive port types $U$ and $U'$ with $s(U), s(U') \subseteq [M_{0}]$ the step relation is defined by

$$U \xrightarrow{m} U' \iff s(U') = \{M' \in [M_{0}] | \exists M \in s(U). \; M \xrightarrow{m,\epsilon} s.M'\}$$

Here $\xrightarrow{m,\epsilon}$ denotes one step labeled with operation $m$ and an arbitrary number of $\epsilon$-steps.
In contrast to state step relation, which is non-deterministic from the usage perspective, the exclusive state set step relation complete in the sense that the resulting state set is unique and contains every single reachable state.

**Shared Usage Ports**

When shared usage ports are considered, we have to take into account the arbitrary effects of sharing. Thus the initial sets has to take into account that another shared port may interact with the provide port. If for a shared standard provide port of a newly created net with type \( P \) the corresponding shared usage port \( U = P \) is determined by,

\[
\delta(U) = \{M' \in [M] | \exists w \in A' \cup \{ \} M \xrightarrow{w} \xrightarrow{\epsilon} M' \}.
\]

**Definition 3.2** For two shared port types \( U \) and \( U' \) with \( \delta(U), \delta(U') \subseteq [M_0] \) the step relation is defined by

\[
U \xrightarrow{m} U' \iff \delta(U') = \{M' \in [M_0] | \exists M \in \delta(U), w \in A' \cup \{ \} M \xrightarrow{w} \xrightarrow{\epsilon} M' \}.
\]

The shared state set step relation is also complete in the sense that the resulting state set is unique and contains every single reachable state even when other shared ports interfere. Thus for all case, the provide side, the exclusive usage and shared usage side an unique step relation is established.

**Consistency**

By demanding that every created pair of ports \((\rho, \phi)\) must be consistent, we can consider rules guaranteeing that only consistent pairs are reachable. But the situation is even more complex due to the message buffers and shared ports.

**Definition 3.3** An exclusive connection is described by an exclusive usage port \( \phi \), a corresponding QUEUE element \((\varphi, w)\) and a providing port \( \rho \).

\[
\phi \leftrightarrow (\varphi, w) \leftrightarrow \rho.
\]

This situation is consistent iff for \( w = m_1 \ldots m_n \rho : P_0 \in P \) and \( \varphi : U_0 \in U \) holds:

\[
\forall i \in [1 : n] \ U_{i-1} \xrightarrow{m_i} U_i \text{ and } \phi : U_n \wedge s(P_0) \in s(U_0).
\]

We have to ensure that the consistency rule concerning the usage and provide port do not exclude not safe interactions like the occurrence of deadlock by definition. Otherwise these kind of phenomena can not be studied in the model.

For our system behavior we have to show that each transformation preserves consistency. For a shared connection the situation is slightly more complex.
Definition 3.4 A shared connection is described by a providing port \( \rho \) connected with multiple usage ports \( \phi^i \) and there exists QUEUE\( \) elements \((\varphi^i, w^i)\) for all of them.

\[
\phi^1 \leftrightarrow (\varphi^1, w^1) \\
\vdots \\
\phi^i \leftrightarrow (\varphi^i, w^i) \rightarrow \rho \\
\vdots \\
\phi^m \leftrightarrow (\varphi^m, w^m)
\]

Such a situation is consistent iff

\( P \) is in a durable state \( \Rightarrow \) for every \( i \in [1 : m] \) the notion of consistency concerning the exclusive connection must be fulfilled

\( P \) is in a temporary state \( \Rightarrow \) there exists exactly one usage port which is consistent concerning the exclusive case.

3.3.3 Port Typing

We will define functions \( \Delta, \nabla \), snd and rcv to describe correct port manipulations and correct port pairs for communication.

Correct Usage

Based on the protocol a corresponding port change function \( \Delta : P \times P_u \times A \mapsto P \) can be defined as

\[
\Delta(\zeta, \phi, op) := \begin{cases} 
\varphi^p \zeta = \varphi^p \land \phi = u^r \land P \xrightarrow{op} P' \\
\varphi^u \zeta = u^r \land U \xrightarrow{op} U'
\end{cases}
\]

For a provide port the step relation determines the resulting protocol state and additionally the last interaction partner is stored changing the second identity number accordingly. An usage port is simply determining the resulting protocol state set using the corresponding step relation for the exclusive or shared case. We will later use \( \Delta \) to ensure that send and receive only occurs in a protocol conform order.

Correct Usage

The \( \Delta \) function describes all possible interactions using the defined state or state set step notions. To ensure that only secure asynchronous interactions take place we use \( \nabla : P \times A \mapsto \text{Bool} \) defined as follows

\[
\nabla(\zeta, op) := \begin{cases} 
\text{true} \quad \zeta = \varphi^p \land \phi \in \text{init}(s(P)) \\
\text{true} \quad \zeta = u^r \land M \in s(U) \land \text{M durable} \Rightarrow op \in \text{init}(M) \\
\text{false} \quad \text{else}
\end{cases}
\]
State Change

To detect if two ports are connected we additionally define a function \( \text{snd} : \mathbb{P} \times \mathbb{P}_u \rightarrow \text{Bool} \) defined by

\[
\text{snd}(\zeta, \phi) := \begin{cases} 
\text{true} & \zeta = u^{x,y}_U \land \phi = u^{x,y}_U, \\
\text{false} & \text{else}
\end{cases}
\]

that evaluates to \( \text{true} \) when \( \zeta \) can send to \( \phi \). A function \( \text{rcv} : \mathbb{P} \times \mathbb{P}_u \rightarrow \text{Bool} \) to ensure that the second port is a correct source to receive from is defined by

\[
\text{rcv}(\zeta, \phi) := \begin{cases} 
\text{true} & \zeta = p^{x,y}_P \land \phi = u^{x,y}_U, \\
\text{false} & \text{else}
\end{cases}
\]

Synchronous Send and Receive

A send and receive can simply be described by locally changing the port type from the old state type to a port with the resulting protocol state type.

\[
\text{(sender)} \quad u^{x,y}_U \xrightarrow{m} u^{x,y}_{U'} = \Delta(u^{x,y}_U, \mathbb{P}_{\text{no}}, m) \quad \rightarrow \quad p^{x,x}_P \xrightarrow{m} p^{x,y}_P = \Delta(p^{x,x}_P, u^{x,y}_U, m) \quad \text{(receiver)}
\]

The receive side is remembering the last request usage port by storing it in its second unique identity number \( y \).

Remote Procedure Call

The remote procedure call case is slightly more complex, because the direction of interaction has to change.

\[
\text{(caller)} \quad u^{x,y}_U \xrightarrow{m} u^{x,y}_{U'} = \Delta(u^{x,y}_U, \mathbb{P}_{\text{no}}, m) \quad \rightarrow \quad p^{x,x}_P \xrightarrow{m} p^{x,y}_P = \Delta(p^{x,x}_P, u^{x,y}_U, m) \quad \text{(callee)}
\]

The stored reply usage port is used to identify the reply destination. We see that the return alternative \( m_i \in \{m_1, \ldots, m_n\} \) may influence the state of the resulting ports \( U'' \) and \( P'' \).

3.3.4 Typing Places and Object Orientation

In an object-oriented system subtype polymorphism is a suitable mechanism to simplify the usage for a set of instances with the same supertype. The usage of supertypes to access subtypes allows to abstract from several subtype details. This mechanism is integrated into
3.3 Coordination Nets

the non object-oriented high level net standard by extending the carrier for corresponding types (sorts) in a way that subtype values are always contained. Thus for any type \( T_2 \) derived from a type \( T_1 \) \((T_2 \rightarrow T_1)\) we assume subtyping and thus \( T_2 \) must be a subtype of \( T_1 \) \((T_2 \leftrightsquigarrow T_1)\) and for the carrier sets \( H_{T_2} \subseteq H_{T_1} \) is required.

3.3.5 Annotations

In the following we will formally introduce the additional annotations needed for coordination nets.

Send and Receive Annotations

Three operators \(!, !_a\) and ? are used to add synchronous send, asynchronous send and receive expressions. For a message word variable \( w \in V^{\text{Msg}}\), an internal step \( \epsilon_{P,P'} \), usage port \( \phi \), ports \( \zeta \) and \( \eta \), provide ports \( \rho \) and \( \varrho \) and a message expression \( \langle m \rangle \in \text{TERM}(\Omega \cup V)^{\text{Msg}} \) we have

\[
\eta = \zeta ? \langle m \rangle / \phi, w \text{ as receive command,}
\]

\[
\eta = \zeta !_s \langle m \rangle / \phi \text{ as synchronous send command,}
\]

\[
\eta = \zeta !_a \langle m \rangle / \phi, w \text{ as asynchronous send command and}
\]

\[
\varrho = \rho - \epsilon_{P,P'} \text{ as internal step on a provide port.}
\]

The expressions can be interpreted using the following equivalences

\[
\eta = \zeta ? \langle m \rangle / \phi, w \quad \equiv \quad \eta = \Delta(\zeta, \phi, m) \land m_* = \langle m \rangle \land \text{rcv}(\zeta, \phi),
\]

\[
\eta = \zeta !_s \langle m \rangle / \phi \quad \equiv \quad \eta = \Delta(\zeta, \phi, m) \land m_* = \langle m \rangle \land \text{snd}(\zeta, \phi),
\]

\[
\eta = \zeta !_a \langle m \rangle / \phi, w \quad \equiv \quad \eta = \Delta(\zeta, \phi, m) \land \nabla(\zeta, \text{op}) \land m_* = \langle m \rangle \land \text{snd}(\zeta, \phi) \text{ and}
\]

\[
\varrho = \rho - \epsilon_{P,P'} \quad \equiv \quad \varrho = \Delta(\rho, \text{P}_{\text{no}}, \epsilon_{P,P'}).\n\]

In the equivalences, the \( \Delta \) function determines the resulting port value, a fresh variables \( m_* \) and \( s_* \) are used to store the message contents and synchronize the state. The functions \( \text{snd} \) and \( \text{rcv} \) ensure that the port \( \zeta \) can send to or receive from \( \phi \).

\[
\eta = \zeta ! \langle m \rangle
\]

The general form of a send with operator ! is used as a short cut for either synchronous or asynchronous. Thus the transition has to be represented by two versions for each case. We demand that at most one general send is annotated to one transition and thus the needed expansion will at most double the number of transitions.
Create Annotations

A new operator \( @ \) is used to describe net creation, the generation of a pair of corresponding provide and usage ports, obtaining an additional shared send port by cloning another shared usage port or reducing an exclusive usage port to a shared one. For \( r, q \) distinct variables of the \text{REVERSE}, \( P \) an exclusive provide port type and \( P' \) a shared one we have

\[
\begin{align*}
@P/r &= (u^r_P, u^{r'}_P), \\
@P'/r, q &= (u^{r'}_P, u^{r q}_P), \\
@N/r &= P^{r r}_{\text{netType}(N)} \text{ and} \\
@\phi/r &= \text{portClone}(\phi, r).
\end{align*}
\]

Thus we can define \( \text{TERM}(\Omega \cup V)_{\text{cmd}} \) as the set of all such terms.

\textbf{Definition 3.5} All terms build by a message word variable \( w \in V^{\text{msg}^*} \), two distinct reverse variables \( r, q \in V^{\text{REVERSE}} \), an internal step \( \epsilon_{P, P'} \), port variables \( \zeta, \eta \in V^P \), usage port variable \( \phi \in V^U \), a message term \( \langle m \rangle \in \text{TERM}(\Omega^b \cup V^b)_{\text{msg}} \), \( P \) an exclusive provide port type, \( P' \) a shared one and a net type \( N \in N \) by all type correct expressions concerning \( \Delta \) and \( \nabla \)

\[
\eta = \zeta^r_U\langle m \rangle/\phi, w, \quad \eta = \zeta^r_U\langle m \rangle/\phi, \quad \eta = \zeta^?_U\langle m \rangle/\phi, w \quad q = \rho - \epsilon_{P, P'}
\]

are building the term sets \( \text{TERM}(\Omega^+ \cup V^+)_{\text{asnd}} \), \( \text{TERM}(\Omega^+ \cup V^+)_{\text{snd}} \) and \( \text{TERM}(\Omega^+ \cup V^+)_{\text{rcv}} \) which can be combined to the set of terms \( \text{TERM}(\Omega^+ \cup V^+)_{\text{com}} \) and \( \text{TERM}(\Omega^+ \cup V^+)_{\text{estep}} \) the set of internal steps. All terms of the form

\[
(\rho, \phi) = @P/r, \quad (\rho, \phi) = @P'/r, q, \quad \phi = @N/r \quad \text{or} \quad \varphi = @\phi/r.
\]

build the term sets \( \text{TERM}(\Omega^+ \cup V^+)_{\text{pnew}} \), \( \text{TERM}(\Omega^+ \cup V^+)_{\text{nnew}} \) and \( \text{TERM}(\Omega^+ \cup V^+)_{\text{pclone}} \) which build together the set \( \text{TERM}(\Omega^+ \cup V^+)_{\text{rev}} \). All commands are united to the set

\[
\text{TERM}(\Omega^+ \cup V^+)_{\text{cmd}} = \text{TERM}(\Omega^+ \cup V^+)_{\text{com}} \cup \text{TERM}(\Omega^+ \cup V^+)_{\text{rev}} \cup \text{TERM}(\Omega^+ \cup V^+)_{\text{estep}}.
\]

We now can build a set of such command terms and restrict it to contain no shared reverse or message word variables. For each single term the reverse variable \( r \) and the message word variable \( w \) can only appear once, because message expressions can not contain any such variables. Thus correct sequences of terms can be specified by excluding their shared occurrence in different terms. Additionally we want to exclude that more than one synchronous send or receive expression is annotated to one transition. Otherwise this can result in synchronizing multiple transitions which results in a too powerful synchronous interaction mechanism.
3.3 Coordination Nets

Definition 3.6 Any sequence \( t_1, \ldots, t_n \) of command terms \( t_i \in \text{TERM}(\Omega \cup V)_{\text{Cmd}} \) where for all \( i \neq j \) holds
\[
\text{free}(t_i) \cap \text{free}(t_j) \cap (V^{\text{REVERSE}} \cup V^{\text{Msg}^*}) = \emptyset
\]
and
\[
|\{t_1, \ldots, t_n\} \cap (\text{TERM}(\Omega \cup V)_{\text{send}} \cup \text{TERM}(\Omega \cup V)_{\text{recv}})| \leq 1
\]
is named correct. We define \( \text{STERM}(\Omega \cup V)_{\text{Cmd}} \) to be the set of all correct sequences of \( \text{TERM}(\Omega \cup V)_{\text{Cmd}} \) terms.

By convention we will omit the reverse, message word and send port variables behind the slash when encoding net annotations and simply use
\[
\eta = \zeta^m, \quad \eta = \zeta^m, \quad \eta = \zeta^m, \quad \varphi = \rho - \epsilon_{P,P'},
\]
\[
(\rho, \varphi) = @P, \quad \phi = @N \text{ or } \varphi = @\phi.
\]

3.3.6 Coordination Net Graph

To define the structure of a single net instance we assume a global consistent place typing function \( C \) for the global set \( S \) of all places and a global set of all transitions \( T \) with \( S \cap T = \emptyset \). The set of all types for such net instance is given by \( \mathcal{N} \). One such net type for a single net instance is defined by the following coordination net graph notion.

Definition 3.7 For a given global place set \( S \), a global transition set \( T \), a typing function \( C \) and common boolean signature \( \Sigma^c = (R^c, \Omega^c) \) with \( \Sigma \)-algebra \( H^c = (R^c_H, \Omega^c_H) \) a coordination net graph (type) \( N \), element of the global net type set \( \mathcal{N} \), is a 5 tuple \( (NG, \Sigma, C, AN, M_0) \) with
\[
\text{NG} = (S, T, F) \text{ is a finite Petri net or net graph as defined in } 2.1 \text{ with } S \subseteq S \text{ and } T \subseteq T.
\]
\[
\Sigma = (R, \Omega, V) \text{ is a boolean signature with } R \supseteq R^c. \text{ It has a corresponding } \Sigma \text{-algebra, } H = (R_H, \Omega_H) \text{ which is an extension of } H^c = (R^c_H, \Omega^c_H) \text{ with } \text{Msg}^*, \text{Id,REVERSE} \notin R_H \text{ and } \Omega \supseteq \Omega^c.\footnote{We exclude this technical carriers here to ensure that they can not occur in the usual annotations.}
\]
\[
C : S \to R_H \equiv C|_S \text{ is the function for typing}\footnote{Imported places with covariant instead of invariant typing results in imported places with sub views that depend on the specified subtype. Thus all elements that are subtypes of the original place type but not of the type of the local place are not visible. Using contravariance conflicts when produce edges are used. Then the inserted elements are supertypes and may not fit to the real place because a supertype provides less features.} \text{ places.}
\]
\[
AN = (A, TC, SL) \text{ is a triple of annotation functions with }
\]
A : \text{F} \rightarrow \text{BTERM}(\text{\Omega} \cup \text{V}) \text{ with } H_\text{r} = C(s) \text{ and for any } (s, t), (t', s) \in F \text{ A}(s, t), A(t', s) \in \text{BTERM}(\text{\Omega} \cup \text{V}) \text{ annotates a multi-set of terms to each edge, }

T \text{C} : T \rightarrow \text{BTERM}(\text{\Omega} \cup \text{V} \cup \text{Bool} \cup \text{STERM}(\text{\Omega}^+ \cup \text{V}^+)) \text{Cmd annotate boolean guard expressions, port creation commands, net creation commands, port cloning commands, send commands, and receive commands to a transition and }

S \text{L} : S \rightarrow \text{Bool separates } S \text{ into imported places } S_{\text{imported}} := \{s \in S \mid SL(s) = \text{true}\} \text{ and local places } S_{\text{local}} := \{s \in S \mid SL(s) = \text{false}\}.

M_0 \text{ is the initial marking. A marking must be a function } M : S_{\text{local}} \mapsto \bigcup_{s \in S_{\text{local}}} \mu C(s) \text{ such that } \forall s \in S_{\text{local}} M(s) \in \mu C(s). M_0 \text{ must be a marking for } N \text{ which is restricted to contain exactly one provide port } (P(N, M_0) = \{p_{\text{std,std}}\} \text{ with } P(N, M) := \{x \in P \mid \exists s \in S \text{ x } \in M(s)\}). \text{ The type of a net } (\text{netType}(N)) \text{ is defined as the type of this standard receive port value } p_{\text{std,std}}.

We have to restrict the net graph NG here\textsuperscript{7} to exclude generator transitions and that a transition of a net specifies behavior that is not related to a specific net instance by demanding that for every } t \in T \text{ holds } S_{\text{local}} \times \{t\} \cap F \neq \emptyset.

For } N \in \mathcal{N} \text{ we use } S^N, T^N \text{ etc. to denote the corresponding elements of a net type } N = ((S^N, T^N, F^N), \ldots).

3.3.7 Syntactical Restrictions

In contrast to literals represent ports unique connections and thus we have to restrict their implicit copying to support their explicit management. Implicit generation of copies by multiple occurrences must be excluded.

To provide suitable syntactical restrictions we have to distinguish usage and binding for port variables in the annotation terms. All occurrences are usually usage with two exceptions.

First the resulting ports of communication or create expressions are bind variables for that terms the other used. Second the cloning of a shared usage port does not add the source port to the used set.

**Definition 3.8** A transition } t \in T \text{ is correct if no port is implicit copied or created. The exclusion of implicit copying can be formalized by demanding that every single term does not contain any variable twice, ports do not occur in multi-set terms with cardinality 2 or more, port constants are not used at all,}

\[
\forall (t, s) \neq (t, s') \in F \quad \forall tr \in A(t, s) \text{ tr}' \in A(t, s') \quad \text{free(tr) } \cap \text{ free(tr')} \cap V^P = \emptyset,
\]

\textsuperscript{7} The restriction depends on the local and imported places and thus can not be considered for basic Petri nets.
which guarantees that a port variable is not implicit copied by multiple occurrences in the post-condition annotations of different edges,

$$\forall (t, s) \in F \quad tr \in TC(t) \quad free(A(t, s)) \cap used(tr) \cap V^F = \emptyset$$

which excludes that used ports of a communication annotation are also occur in the post-condition and

$$\forall tr \neq tr' \in TC(t) \quad used(tr) \cap used(tr') \cap V^F = \emptyset$$

which ensures that ports are not used twice in communication annotations. The implicit creation can be excluded by demanding that all used port variables are determined either by pre-condition edge annotations or bound in transition annotations

$$\bigcup_{(s, t) \in F} free(A(s, t)) \cup bound(TC(t)) \supseteq \bigcup_{(t, s) \in F} free(A(t, s)) \cup free(TC(t)).$$

Another possible restriction is to exclude in general unbounded variables in post-conditions and send expressions. Such unbounded variables combined with the guard expression might be useful to specify system behavior, but are often too powerful, because the existence of an assignment might be not decidable in general.

$$bound(t) := \bigcup_{(s, t) \in F} free(A(s, t)) \cup bound(TC(t))$$

$$used(t) := \bigcup_{(t, s) \in F} free(A(t, s)) \cup used(TC(t))$$

**Definition 3.9** A coordination net system CNS is oracle free if for any $t \in T$ holds that

$$bound(t) \supseteq used(t).$$

An oracle free net is always a high level Petri net, but the standard allows to use unbound variables to specify the semantic by guessing possible solutions in a very powerful but not constructive fashion.

Another useful restriction is to forbid guard expressions, that result in dependencies between different pre-condition arcs.

**Definition 3.10** A coordination net system CNS is simply enabling if for any $t \in T$ holds that pre-condition edges contain no common free variables

$$\forall (s, t), (s', t) \in F \quad free(A(s, t)) \cap free(A(s', t)) = \emptyset,$$

and every guard expression can be evaluated with free variables of a specific incoming edge

$$\forall tr \in TC(t) \cap TERM(\Omega \cup V)_{Boo1} \quad \exists (s, t) \in F \quad free(tr) \subseteq free(A(s, t)).$$
3.3.8 Coordination Net System

In a coordination net system that is build by a dynamically changing set of coordination net graph instances a token marking a place is described by a triple \((id, s, v)\) of a net index \(id\), a place \(s\) and a valid value for it \(v \in \mathcal{C}(s)\). Thus the set of places is defined as

\[
\text{PLACE} = \{(id, s, v) | id \in \mathcal{I}d \land s \in S_{\text{local}}^{\text{IdType}(id)} \land v \in \mathcal{C}(s)\}.
\]

A corresponding multi-set \(\mu\text{PLACE}\) corresponds to the system marking \(\mathcal{M}\). For a location \((id, s)\) with \(id \in \mathcal{I}d_N\) and \(s \in S_{\text{local}}^N\) a system marking \(\mathcal{M} \in \mu\text{PLACE}\) can be restricted to be a marking for an instance of a coordination net graph of type \(N\) with index \(id\) by apply \(id\) \((M = \mathcal{M}(id) \in \bigcup_{s \in S_{\text{local}}} \mu\mathcal{C}(s))\).

But for \(s \in S_{\text{imported}}^N\) we have to determine the net instance containing the real place. We use the ability to locate the parent net index to identify the net containing the original place and thus the marking for the imported place. To keep things simple we restrict place sharing to occur only for identical places \(s \in \mathcal{S}\) and thus a consistent typing for these places is ensured by the global typing function \(\mathcal{C}\).

The location of the corresponding net index is given by the nearest net instance when traveling back in the the creation tree as demonstrated in figure 28. This way some kind of textual scoping concerning the creation can be provided. An instance wide sharing of places using dynamic scoping like in lisp concerning the call stack is not useful to provide a for objects suitable mechanism, because the message passing based interaction does not provide any such structure. Thus the object instance wide sharing of attributes as usual in object-oriented languages for all methods of an object can be realized even when the object behavior is realized using multiple net instances. The net index of the real location can be determined using the recursive function \(\text{idLocate}\) defined as follows

\[
\text{idLocate}(id, s) := \begin{cases} 
  id & s \in S_{\text{imported}}^{\text{IdType}(id)} \\
  \text{idLocate}(\text{idParent}(id), s) & \text{else}
\end{cases}
\]

By ensuring that creation expressions for a subnet of type \(N'\) are only allowed if for every imported places of \(N'\) exists a local one in the creating net or one of its parent nets the termination of \(\text{idLocate}\) can always be guaranteed.
3.4 Non Concurrent Step Semantic

Coordination Net System

How to formally combine a system based on the definition of a coordination net graph (definition 3.7) is described in the following.

**Definition 3.11** A coordination net system CNS is a 8 tuple \((N, S, T, C, \Sigma, M_0, P_0, R_0)\) with

\[ N \text{ is a set } \{N^0, N^1, \ldots \} \text{ of coordination net graph types for } S, T, C, \Sigma \text{ and } N. \]

\[ S \text{ a global place set and } T \text{ a global transition set with } S \cap T = \emptyset. \]

\[ \Sigma^c = (R^c, \Omega^c) \text{ is the common signature without variables and the corresponding } \Sigma\text{-algebra } H^c = (R^c_H, \Omega^c_H). \] It builds the subset that all single net graphs share. Each of them uses an extension of it (see definition 3.7).

\[ M_0 \in \mu\text{PLACE is the initial system marking, } P_0 \in \mu\text{QUEUE with } P_0 \subseteq \text{QUEUE is the initial port marking and } R_0 \in \mu\text{REVERSE with } R_0 \subseteq \text{REVERSE and } \{|r \in \text{REVERSE}| R_0(r) \geq 1\} = \infty \text{ is the infinite initial reverse universe.} \]

For every \(id \in \mathbb{I}_d\) holds \(M_0(id)\) is either empty or a marking for a net type \(N = \text{idType}(id)\) and for all \(\phi \in P_u \cap \{x \exists (id, s, x) \in \text{PLACE } M((id, s, x)) > 0\}\) holds \(\exists \varphi \in P_u \exists w \in \text{Msg}^{*} \varphi \xrightarrow{w} \phi \wedge P_0((\phi, w)) \neq 0\) which ensures that for every known usage port a queue exists and the port connections are consistent.

3.4 Non Concurrent Step Semantic

The usual semantic of every instance of a coordination net graph is like that one for a high level Petri net graph, only when synchronization or communication with other coordination net graphs is performed or port or net creation is considered we have to add the related semantic. We will at first only consider the enabling condition and transition rule for one transition like in the standard, to keep things simple.

Like in the standard an assignment \(\alpha\) for all free variables of a single transition \(t \in T^N\) for a net graph \(N \in N\) is considered. To identify the net instance of concern we additionally assume a given net index \(id \in \mathbb{I}_d_N\). For such a triple \((id, t, \alpha)\) we will define the necessary guard conditions, pre- and post-conditions for each single edge or annotation. Afterwards we will combine them to define enabling and occurrence rules. The synchronous communication mechanism can be describe by consider transition pairs \((id, t, \alpha), (id', t', \alpha')\) and define an additional synchronization condition \(\text{Sync}_P\) that must match for a suitable sender and receiver pair. We will further develop this first simple not concurrent enabling and step semantic by specify a concurrent one in section 3.5.
3.4.1 Valid Assignments and Guards

Definition 3.12 For a net graph $N \in \mathcal{N}$ and a net index $id \in \mathbb{I}d_N$, a transition $t \in T_N$ and an assignment $\alpha$ for $t$ we define $(id, t, \alpha)$ to be a valid transition iff
\[
\forall tr \in TC_N(t) \quad [tr]_{\text{Bool}}^\alpha = \text{true} \quad \land \quad tr \not\in \text{TERM}(\Omega \cup V)_{\text{send}}.
\]
The set of all valid net transitions can be defined as $\text{TRANS}_\text{net}$.

The definition guarantees that every boolean guard evaluates to $\text{true}$, all communication expressions and net and port creation equations are correct and that the transition contains no synchronous send.

3.4.2 Pre-Conditions

For an index $id \in \mathbb{I}d_N$, transition $t \in T_N$ and a valid assignment $\alpha$ we can define the following pre-conditions.

Marking

For the normal edge annotations we have the usual symbolic sum
\[
\text{Pre}_M((id, t, \alpha)) := \sum_{id' = \text{idLocate}(id, s) \land (s, t) \in P_N} id' \times s \times \text{Val}_\alpha(A(s, t)) \quad \in \mu\text{PLACE},
\]
where the net index containing the "real place" is determined using the recursive function $\text{idLocate}$ from page 48.

Asynchronous Communication

A receive uses a guessed buffer contents $w$ and usage port $\phi$, which results in the pre-condition
\[
\text{Pre}_{P,?_a}((id, t, \alpha)) := \sum_{\eta = \zeta((m))/\phi, w \in TC(t)} ([\phi]_{\mathcal{P}_\alpha}, [\langle m \rangle]_{\mathbb{M}_\alpha}; [w]_{\mathbb{M}_\alpha})
\]
Thus a message queue contents starting with a $\langle m \rangle$ is demanded. and for an asynchronous send invocation we can guess the old queue contents $w$ and get
\[
\text{Pre}_{P,!_a}((id, t, \alpha)) := \sum_{\eta = \zeta_w((m))/\phi, w \in TC(t)} ([\phi]_{\mathcal{P}_\alpha}; [w]_{\mathbb{M}_\alpha}) \quad \in \mu\text{QUEUE}.
\]
We combine these single pre-conditions for a single transition and port management to the following one
\[
\text{Pre}_P := \text{Pre}_{P,?_a} + \text{Pre}_{P,!_a}.
\]
3.4 Non Concurrent Step Semantic

Synchronous Receive and Send

If send and receive should occur synchronous the *synchronization condition* and pre-
condition must be defined. For a receive that is involved in a synchronous communication
we get

\[ \text{Sync}_{s}((id, t, \alpha)) := \sum_{\eta=\zeta(m)/\phi,w} (P, [\phi]_{F_{\alpha}}, [\langle m \rangle]_{\text{msg}}) \]

and for *synchronous send* we get

\[ \text{Sync}_{s}((id, t, \alpha)) := \sum_{\eta=\zeta(m)/\phi} (P, [\phi]_{F_{\alpha}}, [\langle m \rangle]_{\text{msg}}) \]

The sender and receiver side offers for all elements of its set of protocol states \( s([i]_{\text{port}}) \) the synchronization condition.

Thus we can define what a valid transition pair is.

**Definition 3.13** A transition pair \((id, t, \alpha),(id', t', \alpha')\) contains of a sender \((id, t, \alpha)\) with
\( id \in \text{Id}_N, t \in T_N, \alpha \) an assignment for \( t \) and a receiver \((id', t', \alpha')\) with \( id' \in \text{Id}_N', t' \in T_N', \alpha' \) an assignment for \( t' \). The pair is valid iff

\[ \forall tr \in TC_N(t) \quad [tr]_{\text{Bool}}^\alpha = \text{true}, \]

\[ \forall tr' \in TC_N(t') \quad [tr']_{\text{Bool}}^{\alpha'} = \text{true} \quad \text{and} \quad \text{Sync}_{s}((id, t, \alpha)) \cap \text{Sync}_{s}((id', t', \alpha')) \neq \emptyset. \]

The set of all valid transition pairs is given by \( \text{TRANS}_{\text{sync}} \).

The pre-condition is that the corresponding buffer for the send port is empty at it will be
provided by the usage side.

\[ \text{Pre}_{s}((id, t, \alpha)) := \sum_{\varphi=\phi} ([\phi]_{F_{\alpha}}, \epsilon) \in \mu\text{QUEUE} \]

\[ \text{Pre}_{s}((id, t, \alpha)) := \sum_{\varphi=\phi?\langle m \rangle/\chi,w} ([\phi]_{F_{\alpha}}, \epsilon) \in \mu\text{QUEUE} \]

Allocation of Elements of the \text{REVERSE}

For a net and port creation, port cloning etc. we get

\[ \text{Pre}_{R}((id, t, \alpha)) := \sum_{tr \in TC(t)} [r]_{\text{REVERSE}} \in \mu\text{REVERSE}. \]
Synchronized Pair

For a valid synchronized pair of transitions $t, t' \in \text{TRANS}_{\text{sync}}$ we get

\[
\text{Pre}_M(t, t') := \text{Pre}_M(t) + \text{Pre}_M(t'),
\]

\[
\text{Pre}_P(t, t') := \text{Pre}_P(t) + \text{Pre}_P(t') + \text{Pre}_{\gamma_0}(t') + \text{Pre}_{\gamma_0}(t').
\]

\[
\text{Pre}_R(t, t') := \text{Pre}_R(t) + \text{Pre}_R(t').
\]

3.4.3 Enabling

**Definition 3.14** A valid triple $x = (id, t, \alpha) \in \text{TRANS}_{\text{net}}$ or a valid pair $x = (t, t') \in \text{TRANS}_{\text{sync}}$ is enabled for given markings $M$, $P$ and $R$ iff it holds

\[
\text{Pre}_M(x) \leq M \land \text{Pre}_P(x) \leq P \land \text{Pre}_R(x) \leq R.
\]

3.4.4 Post-Conditions

Analogous to the specified pre-conditions the needed post-conditions can be specified.

Marking

For a net index $id \in \mathbb{I}_N$, transition $t \in T_N$ and a valid assignment $\alpha$ we can also define the post-condition for the normal edge annotations as follows

\[
\text{Post}_{M,A}((id, t, \alpha)) := \sum_{id' = \text{idLocate}(id, s) \land (t, s) \in F_N} id' \times s \times \text{Val}_\alpha(A(t, s)) \in \mu\text{PLACE}.
\]

Every net create expression leads to an update of $M$ for a new net index by adding the initial marking for that net and substitute a suitable value for the the standard receive port $p_{\text{std, std}}$

\[
\text{Post}_{M,0_N}((id, t, \alpha)) := \sum_{\phi = 0_N / r \in \text{TC}(t) \mid q = [r]_{\text{INVERSE}} \mid \text{idGen}(id, N, q)} id' \times M^N_0[p_{\text{std, std}} / p_{q, q}].
\]

Thus the combined marking post-condition is

\[
\text{Post}_M := \text{Post}_{M,A} + \text{Post}_{M,0_N} \in \mu\text{PLACE}.
\]
Asynchronous Communication

For a receive and send terms we get

\[ \text{Post}_{P;?a} ((id,t,\alpha)) := \sum_{\eta=\zeta\langle m \rangle / \phi, w \in \text{TC}(t)} (\Delta([\phi]^\alpha_{Pu}, P_{no}, m), [w]^\alpha_{\text{msg}}) \in \mu\text{QUEUE}, \]

where \( \Delta \) determines the resulting new queue port and

\[ \text{Post}_{P;!a} ((id,t,\alpha)) := \sum_{\eta=\zeta\langle m \rangle / \phi, w \in \text{TC}(t)} ([\phi]^\alpha_{Pu}, [w]^\alpha_{\text{msg}}, [\langle m \rangle]^\alpha_{\text{msg}}) \in \mu\text{QUEUE}, \]

where the \( m \) is added to the queue.

Synchronous Communication

The empty queue for the actual usage port is always produced by the usage side as post-condition, even if it is the receive side.

\[ \text{Post}_{P;\leq} ((id,t,\alpha)) := \sum_{\varphi=\phi\langle m \rangle / x \in \text{TC}(t)} ([\varphi]^\alpha_{Pu}, \epsilon) \in \mu\text{QUEUE} \text{ and} \]

\[ \text{Post}_{P;?} ((id,t,\alpha)) := \sum_{\varphi=\phi\langle m \rangle / \psi, w \in \text{TC}(t)} ([\varphi]^\alpha_{Pu}, \epsilon) \in \mu\text{QUEUE}. \]

Port Creation

For a net and port creation and cloning we have to add corresponding empty ports for the new created ports.

\[ \text{Post}_{P,\alpha P} ((id,t,\alpha)) := \sum_{(\rho, \phi)=\emptyset P / r \in \text{TC}(t)} ([\phi]^\alpha_{Pu}, \epsilon) \in \mu\text{QUEUE}, \]

\[ \text{Post}_{P,\alpha N} ((id,t,\alpha)) := \sum_{\phi=\emptyset N / r \in \text{TC}(t)} ([\phi]^\alpha_{Pu}, \epsilon) \in \mu\text{QUEUE} \text{ and} \]

\[ \text{Post}_{P,\alpha \phi} ((id,t,\alpha)) := \sum_{\varphi=\emptyset \phi / r \in \text{TC}(t)} ([\varphi]^\alpha_{Pu}, \epsilon) \in \mu\text{QUEUE}. \]

We combine them to

\[ \text{Post}_{P,\alpha} := \text{Post}_{P,\alpha P} + \text{Post}_{P,\alpha N} + \text{Post}_{P,\alpha \phi} \in \mu\text{QUEUE}. \]

Finally we get

\[ \text{Post}_{P} := \text{Post}_{P,?a} + \text{Post}_{P,\leq a} + \text{Post}_{P,\alpha} \in \mu\text{QUEUE}. \]
Synchronized Pair

For a pair \((t, t')\) \(\in\) \(\text{TRANS}_{\text{sync}}\) we get

\[
\text{Post}_{\mathcal{M}}(t, t') := \text{Post}_{\mathcal{M}}(t) + \text{Post}_{\mathcal{M}}(t') \in \mu\text{PLACE} \quad \text{and}
\]

\[
\text{Post}_{\mathcal{P}}(t, t') := \text{Post}_{\mathcal{P},u}(t) + \text{Post}_{\mathcal{P},u}(t') + \text{Post}_{\mathcal{P},a}(t) + \text{Post}_{\mathcal{P},a}(t') + \text{Post}_{\mathcal{P},a}(t') + \text{Post}_{\mathcal{P},a}(t').
\]

3.4.5 Reverse

For the reverse no post-conditions are needed. The usage is restricted to consume or extract elements of this infinite set. The reuse of unique elements is not considered to simplify the specification by omit such details.

3.4.6 Transition Rule

**Definition 3.15** An enabled net transition \(x = (id, t, \alpha) \in \text{TRANS}_{\text{net}}\) with \(id \in \text{Id}_N\), \(t \in T^N\) and \(\alpha\) a valid assignment or an enabled pair of transitions \(x = (t, t') \in T \times T\) can occur and the corresponding markings are defined by

\[
\mathcal{M}' = \mathcal{M} - \text{Pre}_{\mathcal{M}}(x) + \text{Post}_{\mathcal{M}}(x),
\]

\[
\mathcal{P}' = \mathcal{P} - \text{Pre}_{\mathcal{P}}(x) + \text{Post}_{\mathcal{P}}(x) \quad \text{and}
\]

\[
\mathcal{R}' = \mathcal{R} - \text{Pre}_{\mathcal{R}}(x).
\]

3.5 Concurrent Step Semantic

We extend the not concurrent enabling and step semantic of section 3.4 to demonstrate that our non standard extensions like FIFO message queues are well defined for the concurrent case too. We now have to consider sets or even multi-sets of occurring transitions. For the synchronous combination of two transitions via synchronous send and receive we have to consider valid pairs of send and receive transitions. We combine net transitions and synchronized transition pairs to the set of all transitions

\[
\text{TRANS} := \text{TRANS}_{\text{net}} \uplus \text{TRANS}_{\text{sync}}
\]

and build the corresponding multi-set \(\mu\text{TRANS}\). For a given multi-set of transitions \(T_{\mu} \in \mu\text{TRANS}\) we can then consider concurrent enabling and occurrence. We can do this by build the linear extension of the non concurrent pre-conditions and post-conditions.
3.5 Concurrent Step Semantic

3.5.1 Pre Conditions

For a multi-set of transitions \( T_\mu \in \mu \text{TRANS} \) we can define the system marking pre-condition as the sum of the net transition pre-conditions

\[
\text{Pre}_M(T_\mu) := \sum_{x \in T_\mu} T_\mu(x) \text{ Pre}_M(x) \in \mu \text{PLACE}.
\]

Also the linear extension of the non concurrent case is correct for the port marking post-condition

\[
\text{Pre}_P(T_\mu) := \sum_{x \in T_\mu} T_\mu(x) \text{ Pre}_P(x) \in \mu \text{QUEUE}.
\]

For the net transition create pre-condition we can combine the ones for every single net transitions

\[
\text{Pre}_R(T_\mu) := \sum_{x \in T_\mu} T_\mu(x) \text{ Pre}_R(x) \in \mu \text{REVERSE}.
\]

The definition of \( R \) ensures that for any \( r \in \text{REVERSE} \) \( R(r) \leq 1 \) and thus all pre-conditions that are not a set lead to non enabling. This ensures that new introduced net indices or port values are unique by using unique values from the reverse universe.

3.5.2 Enabling

Enabling can be defined based on these pre-conditions.

**Definition 3.16** A given multi-set of transitions \( T_\mu \in \mu \text{TRANS} \) is enabled for \( M \), \( P \) and \( R \) iff

\[
\text{Pre}_M(T_\mu) \leq M \land \text{Pre}_P(T_\mu) \leq P \land \text{Pre}_R(T_\mu) \leq R. \tag{13}
\]

3.5.3 Post Conditions

The system marking post-condition for a multi-set of transitions \( T_\mu \in \mu \text{TRANS} \) can simply be defined as the sum of the net transition pre-conditions.

\[
\text{Post}_M(T_\mu) := \sum_{x \in T_\mu} T_\mu(x) \text{ Post}_M(x) \in \mu \text{PLACE}.
\]

The port marking post-condition is simply also the linear extension of the non concurrent case

\[
\text{Post}_P(T_\mu) := \sum_{x \in T_\mu} T_\mu(x) \text{ Post}_P(x) \in \mu \text{QUEUE}.
\]
3.5.4 Transition Rule

Definition 3.17 If a given multi-set of transitions \( T_\mu \in \mu\TRANS \) is enabled for \( \mathcal{M}, \mathcal{P} \) and \( \mathcal{R} \) it will result in \( \mathcal{M}', \mathcal{P}' \) and \( \mathcal{R}' \) when it occurs defined by

\[
\mathcal{M}' = \mathcal{M} - \text{Pre}_\mathcal{M}(T_\mu) + \text{Post}_\mathcal{M}(T_\mu), \quad (14)
\]

\[
\mathcal{P}' = \mathcal{P} - \text{Pre}_\mathcal{P}(T_\mu) + \text{Post}_\mathcal{P}(T_\mu) \quad \text{and} \quad (15)
\]

\[
\mathcal{R}' = \mathcal{R} - \text{Pre}_\mathcal{R}(T_\mu). \quad (16)
\]

So a concurrent step semantic for a coordination net system is given. This can be mapped to the high level Petri net model as demonstrated in appendix A.

3.5.5 Runs

Like in section 2.1.6, a partial order of transition occurrences can be defined using an occurrence net and an adjusted form of inscription like that used in [KV98].

Definition 3.18 A \( \Sigma \)-inscription for a coordination net system \( \mathcal{CNS} = (\mathcal{N}, \mathcal{S}, \mathcal{T}, \mathcal{C}, \Sigma^c, \mathcal{M}_0, \mathcal{P}_0, \mathcal{R}_0) \) and an occurrence net \( \mathcal{K} = (C, E, F) \) is a pair of functions \( (r_C, r_E) \) with

\[
r_C : C \mapsto \text{PLACE} \cup \text{QUEUE} \quad \text{with} \quad (id, s, a) \mapsto id \in \mathbb{I}d \land s \in S^{\text{idType}(id)}_{\text{local}} \land a \in C(s)
\]

\[
\text{and} \quad (\phi, w) \mapsto \phi \in \mathbb{P}_a \land w \in \text{Msg}^* \quad \text{and} \quad (\phi, w) \mapsto \phi \in \mathbb{P}_a \land w \in \text{Msg}^*
\]

\[
r_E : E \mapsto \text{TRANS} \times \text{\phiREVERSE}.
\]

For \( Q \subseteq C \) we define \( r_C : \phi C \mapsto \mu\text{PLACE} \cup \mu\text{QUEUE} \) by \( r_C(Q)(id, s, a) = |\{ c \in Q | r_C(c) = (id, s, a) \} | \) and \( r_C(Q)(\phi, w) = |\{ c \in Q | r_C(c) = (\phi, w) \} | \).

This inscription combined with an occurrence net is the demanded structure describing a run.

Definition 3.19 A run for a coordination net system \( \mathcal{NS} = (\mathcal{N}, \mathcal{S}, \mathcal{T}, \mathcal{C}, \Sigma^c, \mathcal{M}_0, \mathcal{P}_0, \mathcal{R}_0) \) is an occurrence net \( \mathcal{K} = (C, E, F) \) and a corresponding \( \Sigma \)-inscription \( (r_C, r_E) \) with

\[
r_C(0\mathcal{K}) = \mathcal{M}_0 + \mathcal{P}_0 \quad \text{and}
\]
### 3.5 Concurrent Step Semantic

\( \forall e, e' \in E \) with \( r_E(e) = (x, R) \) and \( r_E(e') = (x', R') \) holds

\[ r_C(e) = \text{Pre}_M(x) + \text{Pre}_P(x), \quad r_C(e') = \text{Post}_M(x) + \text{Post}_P(x), \]

\[ R = \text{Pre}_R(x) \quad \text{and} \quad R \cap R' = \emptyset. \]

In figure 29 a possible run for the asynchronous communication example of figure 11 is presented. The queue symbol is used instead of the usual place symbol, when a queue element is assigned to a place of the occurrence net. The reverse universe can simply be omitted for runs, because it does not restrict the run in any way.

The synchronous case of a communication is shown in figure 30, where the event \( e \) with \( r_E(e) = (((id, t, \alpha), (id', t', \alpha'))), R) \) is visualized using two transitions combined by a dashed rectangle.
3.6 Fairness and Progress

We will treat the fairness aspect here only informally and assume for the transition sets excluding any quiescent transitions of every coordination net graph instance all runs to be fair. Thus a fair scheduling for all net graph instances is guaranteed, but no fair treatment concerning the internal conflicts between single transitions of a net graph instance is guaranteed. This reflects a situation where for the communication subsystem and the concurrency between multiple net instance views some notion of fairness is provided but for a single net instance only progress is guaranteed. Hence, if always at least one not quiescent transition is locally enabled and also infinite often enabled concerning the global behavior, the net guarantees some progress.
4 Extension with Actions and Pools

To achieve a more expressive Petri net formalism, we extend the coordination net approach by introducing action macros for useful or prototypical situations like the already considered remote procedure call. Besides the visual more compact way of representing typical net substructures using more abstract action symbols we can restrict the usage of variables and their declaration to special cases. This simplifies the treatment of nets to a great extend and results in a more suitable notation for software engineering and practical applications than high level Petri nets are.

4.1 Actions

The idea is to use reentrant subnets in a macro like way for a special kind of transition refinements. For a general overview about transition refinement see Brauer at al. [BGV90]. The usual idea is to refine a single transition by a net that preserves the behavior of the transition. Valette [Val79] refines transitions by subnets called block with one initial and one final transition. The block is protected from multiple occurrences of the initial transition before the final transition occurs, by assuming that the refined transition is not 2-enabled for any reachable marking. Thus the net must not be reentrant.

Suzuki and Murata [SM83] generalize this technique and consider the case, where the refined transition is restricted to be at most $k$-enabled. Work which considers also distributed input and output is done by Vogler [Vog87]. He studies a refinement notion depending on the environment of the transition. He demonstrates that only non distributed input is feasible but distributed output can be used when environment independent refinement is considered.

In contrast to all these considerations we need a really reentrant construction, otherwise the parallel occurrence of actions will be limited. See Chehaibar [Che91] for a definition of reentrant behavior for colored Petri nets. We also restrict the behavior preserving character of an action to the effect of the action and its refinement on the places of the net. The restriction of the nets of the high level Petri net standard to exclude inhibitor arcs as defined for example by Christensen and Hansen [CH93] for colored Petri nets allows to cover the non atomic effects of the refining net by a single atomic transition. But this hiding is misleading in the context of system specification and thus we refine always a pair of transitions, where the initial and final one correspond to the initial and final ones of the refinement. We will later introduce a resource pool construction that supports set oriented access and thus supports inhibitor arcs. Thus the refinement is restricted to transition pairs to avoid any semantical problems due to inhibitor arcs.

As demonstrated in figure 31, the incoming or entry part can consume and produce token. The leave part can in contrast only produce tokens. The leave part can only occur after the corresponding entry part has occurred. So called quiescent transitions marked grey can also occur in the action macro. A quiescent entry part means, that the initial action is
Figure 31: The two step action abstraction

**Definition 4.1** An abstract action net is a coordination net graph $N$ with $S = S_{\text{border}} \cup \{s_{\text{local}}\}$ and $T = \{t_{\text{initial}}, t_{\text{final}}^1, \ldots, t_{\text{final}}^n\}$. A set of so called border places $S_{\text{border}}$ and a local place $s_{\text{local}}$, an initial transition and $n \geq 0$ final transitions with $\bullet t_{\text{final}}^i = \{s_{\text{local}}\}$ and $t_{\text{initial}} \supseteq \{s_{\text{local}}\}$ build the structure.

The case $n = 0$ corresponds to an activity that is only initiated. No synchronization with its termination is possible then. For $n > 1$ we have $n$ alternative outputs that may provide different results or indicate different decisions. The intended refinement is described using a net that is allowed to contain an arbitrary set of local places and transitions.

**Definition 4.2** An action refinement net is a reentrant coordination net graph $N$ with an initial transition $t_{\text{initial}}^N$, a set of final transition $t_{\text{final}}^N$ and a set of border places $S_{\text{border}} \subseteq S^N$ with $\bullet t_{\text{final}}^N \cap S_{\text{border}} = \emptyset$. All internal transitions are not connected to the border places.

$$S^N \times \{t_{\text{initial}}^N\} \supseteq F^N \cap S_{\text{border}} \times T^N \quad \{t_{\text{final}}^N, \ldots\} \times S^N \supseteq F^N \cap T^N \times S_{\text{border}}$$

An action declaration is build by a pair of an abstract action pair net and a correct refinement net for it, where the abstract action net behavior contains its refinement.

**Definition 4.3** An action is specified by a pair $(P, N)$ of an abstract action net $P$ and an action refinement net $N$ with the same border place set and for every initial marking and run $(K, (r_C, r_E))$ of $N$ exists a run $(K', (r_C', r_E'))$ of $P$ with

$$r_C(K^0) = r_C'(K'^0) \quad \text{and} \quad r_C(K^0) = r_C'(K'^0).$$

If the initial transition of $N$ is quiescent the initial one of $P$ has to be quiescent, too. If there do exist infinite runs for $N$ or runs which do not reach any final transition this can be covered by using a quiescent final transition for $P$. 
Concerning its local effects on the places of any surrounding net the refinement $N$ behaves like the specification $P$ (see figure 32). The functional behavior of the refinement net must be included in the possibly non deterministic behavior of the specification $P$.

An action is an abstraction of the internal behavior of the corresponding sub net, which specifies the behavior only modulo equivalent place, communication and create activity. It behaves place equivalent to the start and end transition abstraction. For example, the already considered remote procedure call can now be handled in general by using this action notion.

To allow a flexible usage, an action is embedded by mapping the $S_{\text{border}}$ set on places of a surrounding net. Additional pre- or post-conditions for $t_{\text{initial}}$ and additional post-conditions for $t_{\text{final}}$ are allowed, because they do not influence the correctness of the action abstraction. The additionally added edges can also be used to bind variables to the action context, which are then available for selection expressions or produce edge annotations. The internal place is omitted and the initial and final transition are combined to build an action containing an initial part and a final part for the case of $n = 1$ as demonstrated in figure 33.

For multiple final transitions, we combine the several alternative transitions to an action with alternative output in the abstract action net ($n > 1$) as visualized in figure 34.

### 4.1.1 Calls

By using a special application edge and graphical points for each argument in declaration order from the top down to the bottom or left to right if rotated by 90 degree most of the

![Figure 32: Behavior relation for an action](image)

![Figure 33: Embedding of an action](image)
annotations can be avoided (see figure 35). The so build action combining two transitions is initiated only locally by the send transition, so we draw them without any shadow.

Remote Procedure Call

A guaranteed processing is assumed, otherwise the leave transition will not occur definitely and thus must be marked to be quiescent.

The basic interaction on the caller side is presented in figure 36.

Additional alternative return vectors \( m_1, \ldots, m_n \) for an operation \( m \) should be described and can be folded as shown in figure 37.

The general treatment of a remote procedure call like call with input variables \( a_1, \ldots, a_n \) and return vectors \( r_1, \ldots, r_n \) is presented in figure 38. We define this to be the semantic for a call action.
Remote Procedure Call with Asynchronous Reply

The variant with asynchronous reply is slightly more complex. A general treatment of a remote procedure call with asynchronous reply like call with input variables $a_1, \ldots, a_n$ and return vectors $\vec{r}_1, \ldots, \vec{r}_m$ is presented in figure 39 where it’s semantic is defined. Remarkable is the fact that the resulting usage port $\psi$ allows no interaction anymore, and thus we can assume that it is not further used. This is reflected by the fact that no edge propagating it is present in figure 39, left-hand-side.
4.1.2 One Way Call

A call without return values is simply a send transition. But to have a representation, which is conform with the remote procedure call, we also introduce explicitly an one way call action for it as defined by the net in figure 40.

4.1.3 Net Creation and Invocation

If a net should be temporarily generated and called directly, this can be done by the net creation and invocation action defined in figure 41. The differences between a call action and a direct net invocation action are that a direct net invocation implies an explicit net creation. A call uses a message to signal the serving net to process the request instead. How the request is processed, is then determined by the serving net and not by the caller.

4.1.4 Call Forward

Until now the described interaction is based on interfaces as typed ports including remote procedure calls and state based protocols. This establishes a certain interface oriented view upon a system. The dynamic creation of ports allows dynamically evolving systems, but remote procedure call like interactions are still processed in a message passing style by one
coordination net graph. A net can use other possibly dynamically created nets to process a request. For implementation purposes this kind of invocation forward is like a "method" call in programming languages. By forwarding a request to a dynamically created net a dynamic created context like a stack frame is provided. See figure 42 for the definition of the *call forward action*. Additional resources needed exclusive by the created net are annotated and send to it together with the received parameters. The short cut for this net invocation action is drawn with a shadow, because all related transitions are depending on the environment and receive messages. This net invocation seems to be a suitable mechanism in order to modularize the nets without introducing additional ports when using the action symbols.

![Diagram](image)

**Figure 42:** A *call forward action* to dynamically generate nets

A version for a request with asynchronous reply is presented in figure 43. The reply is sent to port $\sigma$ provided as parameter, whereas the resulting provide port of the request receiving $\rho$ is directly available after the receive is processed.

In the case of an *one way call*, we have to modify the version for a request with asynchronous reply as presented in figure 44. The request receiving port $\rho$ is directly available after the receive is processed.

![Diagram](image)

**Figure 43:** A *call forward action* for asynchronous reply

4.2 Pools

In a distributed environment, resources are often used in a set oriented way. Although certain elements of such sets may be individually accessed, often only an appropriate chosen
element or subset of such a set should do a certain job. So far, our approach already allow some kind of grouping using places to handle objects or literals, but multi-term annotated edges cannot express, for example, that all elements of a place should be consumed.

4.2.1 Event Pools

The simple case of a place providing consumable elements with event character is covered by adding a symbol for an event pool which permits true concurrent consume and produce access, but does not support set oriented access. To simplify the usage and exclude global effects on the basic control flow event pools are restricted to local places.

4.2.2 Resource Pools

For set oriented access to more durable elements with resource character, a resource pool element is provided. It is used to represent associated objects or attributes which build the essential structure of an object whereas events provide the control flow and dynamics.

To allow such a set oriented access to these places we have to encode the elements. To support a distinction between lock/release and consume/produce access, the elements are combined with identity values from the REVERSE and a set containing all these identity values is stored in an additional place. All related transitions must be extended according to the following rules.

Figure 45 and 46 shows, that a produce arc (or post-condition) inserts a resource \( x \) with a new identity value \( r \) and has to update the set accordingly. When a token is consumed (pre-condition) the identity \( r \) of the chosen element \( x \) of the normal place must be deleted from the set, as defined in figure 46.
4.2 Pools

\[ M(x) = f(M(x)) \]

\[ M - f(M(x)) \]

Figure 45: Expand rule for producing resources

\[ M(x) = M'(x) \]

\[ M - \{ (r, x) \} \]

Figure 46: Expand rule for consuming resources

Simple Selection

The simple selection mechanism presented in figure 47 and 48 can be incorporated with atomic transitions will only be possible for literals and ports concerning the state set. The selection for them is demonstrated in figure 47, where condition \( B \) is applied to a single element \( m[B] \) and the state set is restricted to \( \{ s \} \).

For the limited case where a port is used for synchronous send communication the state restriction can be specified as presented in figure 48. The semantic for synchronous send ensures that sender and receive side guarantee matching states.

Resource Pool Views

For every resource of a local context only one place can exist, but we prefer to split up this place into different disjunct views depending on the requirements of the specified net.

The semantic is presented in figure 49 where each resource view is replaced by access to the real place and the implied restrictions to the consume, produce or read arc are added.

\[ m[B] \]

\[ m \]

Figure 47: Atomic selection with condition \([B] : X \to \text{Bool}\)
Supertype Resource Pool Views

The \textit{polymorphic} treatment of context attributes or associations is one advantage object-orientation offers. We can support this in a flexible manner by allowing to use resource pools that represent a set of different attributes or associations using their common supertype. Due to the read and write semantic of consume or produce edges in general only invariant typing is possible. By restricting such combined resource pools to consume or lock/release usage, also a supertype can be used.

The resulting semantic is presented in figure 50 where each single usage or combined usage is expanded to all possible combination of usages.
Local and Imported Resource Pools

For resources we can distinguish if the underlying place is local or imported. In the case of a simple hexagon the resource represents an element of the net itself and thus is a local place (see figure 51) is used.

A double hexagon is used to represent resource pools for element of the corresponding net context. The semantic is mapped to an imported place (see figure 51).

4.2.3 Locking and Releasing Resources

A first simple form of locking occurs when a two step action uses an element of a local or imported resource.

See figure 53 for the related situations where the possible exchange of a resource element is interpreted as locking and releasing the element. It can be distinguished from a consume or produce by the fact that the action guarantees to leave the number of resources unchanged. The distinction is thus only relevant for selection or replication.

Two usual cases where this form of lock and release occurs are call forward actions that allocate the needed number of resources before starting the service net and release them afterwards. These resources are then represented in the service net using local resource
pools with the name of the original resource pool.

To describe the dynamic locking and releasing of imported resources in a service net we represent the locked elements using an additional local resource pool with identical name.

![Figure 54: Lock a imported resource element](image)

To specify a locking one simply has to transfer an element from the imported pool to that one for exclusive or locked instances. This can be done using an atomic transition or the initial part of an action or by an usual two step action as demonstrated in figure 54 right-hand-side.

![Figure 55: Release a locked imported resource element](image)

The locked resource can be released in the same way by putting the corresponding contract port back into the imported resource pool. For the resources locked manually we have to guarantee that they are released manually after a finite number of steps and before the net terminates; otherwise the lock/release semantic is not respected.

### 4.2.4 Lock Semantic

![Figure 56: Expand rule for locking resources](image)
4.3 Selection and Replication

A lock results simply in a consume of an element (see figure 56) which is put back when the corresponding release takes place (see figure 57). Locking and releasing can be done in a true concurrent way while produce and consume are in conflict, because always only one transition can consume the identity value set at the same time. Even when read-arcs like defined by Montanari and Rossi in [MR95] are considered the sequential nature and the fact that also writes and changes on the resources occur allow no true concurrent solution.

![Diagram of lock and release](image)

Figure 57: Expand rule for releasing resources

4.3 Selection and Replication

In general, set oriented access is usually provided by some kind of queries. Such queries are provided in relational databases using SQL expression [SKS97] and for object-oriented databases its extension OQL [CB97] is used. In contrast to OQL, where the whole class structure can be traversed using path expressions, we restrict such selections to a single set of associations to avoid any complex dependencies between class structures and the behavior specification, because otherwise the contract abstraction would be violated. Thus, we provide constructs for selection, for all access and set selection for the sets of resources. These mechanisms can be combined with two forms of replication that exploit the given sets by either providing a parallel execution for all selected elements (parallel replication) or builds some form of sequential processing pipeline (sequential replication). The query processing is not a priori specified with the common ACID properties [SKS97] of database transactions. Instead a concurrent processing scheme is defined that can be later restricted to exclude all cases for which the ACID properties do not hold for the set of transitions belonging to an action.

4.3.1 Complex Action Scheme

For all actions with two step semantic (which excludes, for example, one way calls) we can use several mixtures of usual pre-condition edges, non-enabling selection, replication edges or set replication edges. A set of examples for the possible combinations is presented in figure 58, where different cases are visualized. To provide a semantic for such complex combined cases we have to describe the semantic for each edge type in a fashion that allows to combine them to obtain the resulting overall semantic.
A scheme that visualizes the semantic for a generalized edges is shown in figure 59. Besides the "init" step that starts the complex action processing an usual termination or abort is described by the "term" and "abort" transition. The "reset" transition and extended "get" and "usage" actions (here we permit precondition for the termination transition) describe the internal processing. When selection or replication is done the basic atomic initial step and termination semantic for actions has to be extended. The allocation and selection of resource is processed in parallel and thus deadlocks may result. A common record is used as data structure to encode the synchronized interleaving of all the processing needed for each different edge. The different parts are characterized in the following subsections.

**Init, Get, Usage, Abort, Term and Reset Part**

During the atomic initial step of the complex action, for each edge the "init" part is performed. For an usual edge or set replication edge, the precondition is simply consumed or locked and the element, sequence or set is inserted into the common record. For a selection or selection replication edge, the set of resource identity values \( R \) is copied twice and inserted together with an empty selection set or sequence \( S \) of resource identity values and resource value pairs and a counter \( f \) with initial value 0.
Then the get part has to be considered. It is processed in parallel only for selection or 
selection replication by locking resources \((r, x)\) with \(r\) in the copied resource identity value 
set. In the initial part of the get action, the identity value \(r\) is erased from the corresponding 
set. For a selection expression annotated to a non-enabling edge, an implicit subnet is 
derived that evaluates if one particular element fulfills the condition. This implicit defined 
subnet is used to determine if the chosen element \(x\) is fulfilling the condition. The pair 
\((r, x)\) is inserted or appended to \(S\) iff the test succeeds and the counter is increased.

For the usage either the element or the first element of the set or sequence is chosen for an 
usual edge or a set replication edge or the resource \(x\) of the first element of the selection 
set \(S\) is used. The locked resources are afterwards returned to their resource pool and 
erased from \(S\). In the case of a set replication the used element is also erased from \(S\). For 
the parallel replication case the elements provided by usual edges must be cloned for each 
anction call and in the sequential case the results are used as input for the next usage.

An abort can only occur for set replication or any kind of selection when the given set or 
selection is empty. This can be determined for a given set by testing \(S = \emptyset \wedge f = 0\) and 
for a selection or selection replication by the condition \(R \cap R' = \emptyset \wedge S = \emptyset \wedge f = 0\) where 
\(R'\) is the actual resource identity value set.

The termination can be detected analogously for a given set by \(S = \emptyset \wedge f > 0\) and for 
a selection or selection replication by \(R \cap R' = \emptyset \wedge S = \emptyset \wedge f > 0\) where \(R'\) again is 
the corresponding actual resource identity value set. For an ordered set \(\{e_1, \ldots, e_n\}, <\), 
the action is terminated, when all its edges are. When a replication edge with less than 
maximal degree is terminated, all smaller edges w.r.t. \(<\) have to be reseted.

How to combine these different steps to obtain the semantic for a complex action is demon-
strated in figure 60. For the given set \(\{e_1, \ldots, e_n\}\) of edges we assume terms \(\text{init}, \text{abort}\) 
and \(\text{term}\) as well as their get and usage actions to combine them to obtain the semantic 
for the complex action. An additional element \(u\) at the end of the common record is used 
to store the number of active usage actions. As long as at least one is active, no reset, 
terminate or abort transition is allowed to change the action processing. We specify the
mentioned terms in the following subsection for each edge type in detail.

### 4.3.2 Usual Edge

For an usual edge $e$, a value $x$ is stored in the common record and we define

$$
\begin{align*}
\text{init}(e) &= v \quad \text{for } v \text{ the bound variable of } e \\
\text{reset}(e) &= x \\
\text{abort}(e) &= \text{false} \\
\text{term}(e) &= \text{true}.
\end{align*}
$$

We assume that at least one edge is not an usual edge and thus omit any additional encoding to ensure that at least one usage action has taken place before the term condition becomes true.

![Diagram of get and usage for a normal edge](image)

A normal edge does not provide any get action and provides the argument either by cloning for the case of a replication or directly as specified in figure 61.

### 4.3.3 Selection Edge

A selection edge adds a 4-tuple $\langle R_o, R, S, f \rangle$ to the common record and the specific terms for an edge $e$ are defined for the current identity value set $R'$ of the resource pool of $e$ as

$$
\begin{align*}
\text{init}(e) &= \langle R', R', \emptyset, 0 \rangle \\
\text{reset}(e) &= \langle R_o, R_o, \emptyset, 0 \rangle \\
\text{abort}(e) &= R \cap R' = \emptyset \land S = \emptyset \land f = 0 \\
\text{term}(e) &= f > 0.
\end{align*}
$$

The selection can not add any further candidates when the initial copied identity value set does not contain some still existing members. Thus, if $f = 0$ holds, the complex action
must be aborted, because the selection is empty. Successful tested candidates are stored in \( S \) and \( f \) is increased when they are used. So for \( f > 0 \), the action has to terminate.

![Diagram 62: Get and usage for a selection edge](image)

The get action of a selection tries to lock an element still stored in the second initially copied set of identity values. If one element is locked the candidate \( x \) is tested. If the test succeeds, the pair \((r, x)\) is inserted into \( S \). Always the corresponding identity is erased from \( R \) and thus the processing terminates at least when all \( r \in R \) are tested. For the usage action, an element of \( S \) is used, the active usage action counter is incremented and decremented, the resource is released and the result is added to the common record.

4.3.4 Parallel Replication Edge

Also a 4-tuple \( (R_0, R, S, f) \) is added to the common record for a parallel replication edge. The needed terms for an edge \( e \) with current identity value set \( R' \) of the resource pool of \( e \) are only distinct from the selection case for the termination

\[
\text{term}(e) = R \cap R' = \emptyset \land S = \emptyset \land f > 0.
\]

![Diagram 63: Get for a parallel replication edge](image)

The get action works like the one for a selection edge with the important difference that
the condition \( S = \emptyset \) is omitted. Thus, the get action of a parallel replication edge is like that one for a selection with the distinction that the get processing is not already stopping when one candidate has been found, but only when all have been tested. The usage action remains the same as in figure 62.

4.3.5 Sequential Replication Edge

We have to add nearly the same 4-tuple \( \langle R_0, R, S, f \rangle \) to the common record for a sequential replication edge. This time in contrast \( S \) has to be a word of pairs. The needed terms for the edge \( e \) are also nearly the same as for a parallel replication edge. We simply have to adjust the usage of \( S \).

\[
\begin{align*}
\text{abort}(e) &= R \cap R' = \emptyset \land S = \epsilon \land f = 0 \\
\text{term}(e) &= R \cap R' = \emptyset \land S = \epsilon \land f > 0.
\end{align*}
\]

\[\text{Figure 64: Get and usage for a sequential replication edge}\]

The get action is like that one for a parallel replication with the distinction that the get processing is appending the tested candidates. The usage is like that of a parallel replication edge with the difference that the first element is extracted and replaced by a bottom pair \((\perp, \perp)\) in order to block the processing and to ensure the pipelining.

4.3.6 Example

See the sequential replication of figure 65 as an example for the presented processing scheme. The initially given time will usually be 0 and thus for a situation where \textbf{servers} contain three server resources \( r_1, r_2 \) and \( r_3 \) we get the initial situation presented in table 1 row 1.

The in table 1 row by row presented processing is one possible observation. First the element \((r_1, x_1)\) is consumed from the resource \textbf{servers}. In contrast to the case of an
Figure 65: Sequential replication to accumulate down time for a set of servers

Table 1: Processing of addDownTime sequential replication

<table>
<thead>
<tr>
<th>activity</th>
<th>$R_0$</th>
<th>$R$</th>
<th>$S$</th>
<th>$f$</th>
<th>term</th>
<th>$t$</th>
<th>term</th>
<th>$u$</th>
</tr>
</thead>
<tbody>
<tr>
<td>get $(r_1, x_1)$</td>
<td>${r_1, r_2, r_3}$</td>
<td>${r_1, r_2, r_3}$</td>
<td>$\epsilon$</td>
<td>0</td>
<td>false</td>
<td>0</td>
<td>true</td>
<td>0</td>
</tr>
<tr>
<td>usage call</td>
<td>${r_1, r_2, r_3}$</td>
<td>${r_2, r_3}$</td>
<td>$(r_1, x_1)$</td>
<td>0</td>
<td>false</td>
<td>0</td>
<td>true</td>
<td>0</td>
</tr>
<tr>
<td>get $(r_2, x_2)$</td>
<td>${r_1, r_2, r_3}$</td>
<td>${r_3}$</td>
<td>$\bot, \bot$</td>
<td>0</td>
<td>false</td>
<td>$\bot$</td>
<td>true</td>
<td>1</td>
</tr>
<tr>
<td>usage return</td>
<td>${r_1, r_2, r_3}$</td>
<td>${r_3}$</td>
<td>$(r_2, x_2)$</td>
<td>1</td>
<td>false</td>
<td>$t_1$</td>
<td>true</td>
<td>0</td>
</tr>
<tr>
<td>get $(r_3, x_3)$</td>
<td>${r_1, r_2, r_3}$</td>
<td>$\emptyset$</td>
<td>$(r_2, x_2); (r_3, x_3)$</td>
<td>1</td>
<td>false</td>
<td>$t_1$</td>
<td>true</td>
<td>0</td>
</tr>
<tr>
<td>usage call</td>
<td>${r_1, r_2, r_3}$</td>
<td>$\emptyset$</td>
<td>$(\bot, \bot); (r_3, x_3)$</td>
<td>1</td>
<td>false</td>
<td>$\bot$</td>
<td>true</td>
<td>1</td>
</tr>
<tr>
<td>usage return</td>
<td>${r_1, r_2, r_3}$</td>
<td>$\emptyset$</td>
<td>$(r_3, x_3)$</td>
<td>2</td>
<td>false</td>
<td>$t_1 + t_2$</td>
<td>true</td>
<td>0</td>
</tr>
<tr>
<td>usage call</td>
<td>${r_1, r_2, r_3}$</td>
<td>$\emptyset$</td>
<td>$(\bot, \bot)$</td>
<td>2</td>
<td>false</td>
<td>$\bot$</td>
<td>true</td>
<td>1</td>
</tr>
<tr>
<td>usage return</td>
<td>${r_1, r_2, r_3}$</td>
<td>$\emptyset$</td>
<td>$\epsilon$</td>
<td>3</td>
<td>true</td>
<td>$t_1 + t_2 + t_3$</td>
<td>true</td>
<td>0</td>
</tr>
</tbody>
</table>

selection expression this time the test if the element fulfills the condition can be omitted and a atomic ”get” behavior can be assumed. After a first candidate has be added to $S$ the ”usage” part can execute it. Thus, ”get” and ”usage” part may execute in parallel. In the example we present a overlapping where $(r_2, x_2)$ are allocated during an active ”usage” for $(r_1, x_1)$. After the first usage returns the last server is allocated $(r_3, x_3))$ and afterwards all servers are processed in the allocated order. For the case of a sequential replication the ”usage” excludes any parallel processing by add a prefix $(\bot, \bot)$ to $S$ while active. All edges are finally terminated and thus the whole action terminates.

4.4 Net Creation

Often a distinct task like a request or a background activity should be described in some kind of isolation using a single net. The two usual situations are the creation of a net for a specific finite task or the creation of a net building a durable entity of the system.

4.4.1 Single Task Net

When for example a request is forwarded to a special single task coordination net to process it or a single asynchronous task should be initiated, often several resources of the
forwarding or initiating net might be needed by the created net and are provided using the import places of the new net. To simplify the parameter and return parameter handling and to represent the single task structure of this created nets, we add special input/output bars representing the invocation at the beginning of the net and the return at its end. The bars represent transitions that initialize and terminate the coordination net. There mapping into usual coordination nets is demonstrated in figure 66. There is a special receive transition for initializing the coordination net and receiving the arguments plus additional exclusive parameters. The termination of the net is done by a special send transition that is used to propagate the results and return the additional exclusive resources. Thus, the net guarantees that a reply is generated only once, but no guarantee that all processing is finished is considered. Even the termination has to be ensured for such a net, because otherwise the remote procedure call contract will not be a correct abstraction. The folding shown in figure 67 abstract from these details.

4.4.2 Creation of Subsystems

When a durable net should be created this can also be done using an action. A factory based approach building some kind of standard factories instead of this pseudo-action is an interesting design alternative, but the provided basic creation action is a more flexible solution. It can simply be used in a restricted fashion to build a factory based design. For the action only the net type $N$ has to be specified. We further assume that other kinds of objects like exceptions or literals of dynamic size can also be created using the same kind of action but without building a corresponding net structure. For a creating net with id $i$ and net graph $N' = ((S', T', F'), \ldots)$ and a $\text{new}N(\ldots)$ action with $N$ net graph $((S, T, F, \ldots)$

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure66}
\caption{Net structure}
\end{figure}
4.5 Conventions

In this subsection we will describe several conventions and shortcuts covering edge types, casts and exceptions.

4.5.1 Casts

For types $T_1$ and $T_2$ with $T_2$ a subtype of $T_1$ ($T_2 \subseteq T_1$), an explicit cast can be specified using a cast expression at a consume or produce edge. Such a cast expression can transform a port type to another sub or supertype. Casts to the supertype are always secure, but

the condition $S' \supseteq S_{\text{imported}}$ must hold.

4.5 Conventions

In this subsection we will describe several conventions and shortcuts covering edge types, casts and exceptions.
casts to a subtype might fail and thus lead to insecure behavior which results in a possible invalid cast exception.

4.5.2 Activation and Parameter Edges

![Activation and parameter edges](image)

As already used in various action definitions, an activation edge is used to assign the port which is the provided or used one. A set of parameter edges and graphical ports denoted by empty cycles for input parameters and filled ones for output parameters are used to describe the parameter assignment (see figure 70). Additional parameter edges describe usual edges that can be annotated with multi-sets of terms to describe consumed elements. The variables of the terms of consume and read edges can then be used in the guard and command expressions or produce edges.

Non Enabling Edges

![Non enabling edge examples](image)

A special form of non enabling edges is supported to describe selection or replication activities (see figure 71). A non enabling activation edge is used to denote that an action is occurring even when the described selection is empty. For all non enabling parameter edges, the evaluation is delayed until the other enabling pre-conditions are fulfilled. Afterwards, when the action is started, the non enabling parameters are consumed. If this is not possible, a resources not available exception is thrown by the action.

4.5.3 Exception Handling

Consider for example an extended CORBA IDL specifying that an operation op can result in two different exceptions Overflow and Underflow. Both exceptions are sub-types of the
4.5 Conventions

OutOfBounds exception$^8$.

```plaintext
exception OutOfBounds { ... };

exception Overflow : OutOfBounds { ... };
exception Underflow : OutOfBounds { ... };

int op(...) raises (Overflow, Underflow);
```

This is the same as the following interface using our multiple output notation:

```plaintext
exception Overflow : OutOfBounds { ... };
exception Underflow : OutOfBounds { ... };

op(...) -> (int), (Overflow), (Underflow);
```

Nevertheless, it is not useful to visualize such exceptions, which are not part of the usual control flow and behavior in the same way as the regular output alternatives. So a special edge for that purpose is introduced. A reaction to any kind of OutOfBounds exceptions may be described as follows:

![Figure 72: The exception edge abstraction](image)

We simply consider exceptions to be special alternative outputs of a call which return only one parameter of a sub-type of `Exception` (see figure 72). Exceptions that are not covered by any exception-arcs lead to an abort, like not catched exceptions in programming languages$^9$. The caught exception types are assigned to the arc implicitly by considering the corresponding pool. This implicit mapping is only correct and unique if all connected exception types are different. If several exceptions are in any subtype relation, always the smallest fitting type concerning the subtype relation is chosen.

An exception is simply thrown in a single task net like any other return alternative by providing an additional part in the output bar as demonstrated in figure 73. One possible

---

$^8$exception in CORBA do not have inheritance like records (struct)

$^9$the explicit propagation of exceptions is difficult due to the multiple parallel activities in a net
scenario is to direct propagate the exception generated by a call to that extra return alternative as demonstrated on the left. A caught exception can also propagated after needed synchronizations have been made or a new one can be generated using a \textit{new action} (right side).

**Figure 73:** Throw an exception
5 Object Coordination Nets

Starting with the Petri net notion [BRRe87], there are only a few principal steps of further refining the general concepts in order to obtain a net model which fits into the object-oriented world for, e.g., specifying a single service in the context of a class. We assume the reader to be also familiar with the basic concepts of classes, methods and inheritance used in any object-oriented language like, e.g., in Java [AG96] to pick a popular example. Object Coordination Nets or for short OCoNs are a specially adjusted Petri net based formalism [GGW98, WGG97]. Its semantic is based on the coordination net formalism specified in section 3 and its action and pool macro extensions of section 4.

5.1 Design Decisions

OCoNs are Petri nets adjusted to fit to object-orientation. Resource pools are used to treat associations represented by ports and actions are used to express the interaction. We integrate Petri nets into an object-oriented structural view. The basic idea is to use the already defined shortcuts and actions of the coordination net approach, a given structural design and combine both in a systematic fashion to specify object systems with coordination nets. The coordination net formalism already provides an interface type based system with an inheritance notion for interfaces and object types, but for implementing the system the notion of class is useful, too.

5.1.1 Classes

An usual approach to allow code reuse in concurrent object-oriented class designs without producing inheritance anomalies is to separate synchronization and processing. For a theoretical explanation why such handling allows to avoid inheritance anomalies see Crnogorac et al. [CRR98]. We try to follow this guideline by assuming for a class the following bus like structure (see figure 74).

The "bus" (resource allocation net or RAN for short) implements the object wide synchronization and the "bus-cards" (service nets or SN) should describe the interaction with external resources. But a perfect isolation is not possible and practical, because the synchronization may depend on processed results. Our approach describes only the abstract coordination aspect of a class with nets and can help to delay the implementation of sequential processing tasks to a later stage of the design process. So at least these sequential processing parts can be reused, even if the synchronization nets must be replaced or adjusted to a great extent. The connections or handles to other instances of a class build the set of contracts an object might be interacting with. They are typed using interface typing and protocol nets (PN). The assumption that every interaction is separated by a protocol net and a corresponding syntactical interface description leads to the strong degree of separation wanted. We can allow to connect classes directly, but this is only a notational
simplification for classes which implicitly specify an unique interface and protocol type. W.r.t. a single action, two types of connected places, namely event pools and resource pools are distinguished. The former category can be interpreted as representing the resources needed to fire an action which may be changed during that activity but tend to have a longer lifetime than a single activation. The latter represent most of the detailed flow of control and the objects which are to be processed and, hence, may be consumed during execution. Because a service may need additional resources besides the unique single carrier of activity, and an object may be used as a parameter in one action, but as a resource in another one, we partitioning edges into parameter edges for parameters/results and the activation edge.

We distinguish three views of behavioral aspects for specifying classes and their abstract interfaces/contracts in a distributed system. For each of the views we use one of the OCoN classes described so far, to model the dynamic aspects on the level of detail best suited for the view at hand:

- **Protocol nets** are used to describe the externally visible interfaces/contracts. The services accessible from the outside as well as the externally visible aspects of the state of instantiations of the class w.r.t. the availability of a service are specified.

- A **resource allocation net** specifies the object-wide coordination. This includes the
request processing and the overall resource usage of the external as well as internal services of an object.

- Separated service nets are used to describe a single service within its class context. It contains the detailed implementation of a service concerning the resource coordination. If coordination is trivial a textually coded method may be used instead.

Hence, a class looks like illustrated in figure 75, where all variants of visual representations are OCoNs.

### 5.1.2 Resource Allocation Net

The resource allocation net should be the only net that deals with the processing of incoming requests. For every operation call it should usually invoke a service net. This invocation forwards the parameters to the new created service net and additionally exports the exclusive needed part of the object environment. Usually, the provided protocol nets are subnets of the resource allocation net, but in general the RAN must only ensure that it serves all its contracts as specified. The structural properties like association, aggregation, composition and attributes are all represented by resource pools as replacement for program variables. One main difference to the representation of a structural object model in programming languages is that resource pools provide a set oriented view on associations by default and
thus multiple associations need no additional usage of technical classes to represent them.

The resource allocation net represents the class internal view of its local resources as well as its external ones highlighted using imported resources with double border for them. Additionally, the provided contracts are represented using resource pools containing the provide ports (skeletons) for each connection. Note, that a resource must always be in a single state iff present at all.

Additionally, the inner activity of an object can be specified using the usual actions. An observer (contract user) might react on the observable spontaneous change to a new state of a contract by using a simple pre-condition edge connected with a pool representing this new state. Thus using call-backs that may result in race conditions in the client behavior can be avoided and we still can use a simple unidirectional contract.

5.1.3 Service Nets

Service nets are restricted to use the single task net input and output bars from section 4.4.1. They can be considered to be like methods in object-oriented programming languages and allow to separate request handling and additional internal tasks from each other. If resources are allocated dynamically, this can of course lead to conflicts between different running service nets or the related resource allocation net.

The number of incoming/outgoing arcs is visualized in the service net by bars for in/call and out/return. A single service specifies via its signature what kind and number of resources it needs exclusive. This supports the concept of locality by avoiding the detailed treatment of complex resource interactions in each of the services. Nevertheless, a service net requires the most advanced visual specification level in our approach because it may be as detailed in control and data flow as a method written in the target programming language.

A principal design decision is that all services are concurrent a priori, i.e. as long as not stated otherwise, a service of a class may be called concurrently with itself as well as with other services of the same class. This does not imply that all calls really work in parallel because that may depend on the availability of resources. In our design, the aggregated elements of a class are interpreted to be resources which enable a class to implement its services. So, a parallel call of different services may be sequentialized due to conflicting resource requests or not.

5.1.4 Protocol Nets

Protocol nets and interface declarations are used to separated the different subsystems and classes from each other. They specify the allowed usage of an interface and thus should be as abstract as possible to avoid publishing any details not relevant for their usage.

A contract port represents a connection to another instance and can also be used to exclude
several synchronization problems by assuming a suitable handling by a proxy and skeleton that encapsulates these problems. If after a state change only a specific information can be requested, patterns like smart-proxies can be applied to avoid overhead. But for our purpose to design the system these technical aspects can be neglected and we simply specify autonomous state changes instead of call-backs to avoid splitting up contracts.

The visible states of a contract can be considered as guard conditions (see Lea [Lea97]) that can be used to request services only when accessible. Thus error-prone designs with protocols where service calls may be blocked until a certain guard condition visible only internally becomes true are avoided.

We restrict protocol nets to finite state machines, but in contrast to state charts [Har88] which are used in the UML [Rat97] to specify the external protocol, extended features like and/or state decomposition are not supported to ensure that the resulting descriptions are still simple enough to allow a seamless embedding into other behavioral specifications.

To handle complex cases, the usage of multiple contracts (c.f. [WBWW90]) in a style supporting separation of concern as, e.g., role based modeling (see [Ree96]), is applied.

5.1.5 Advantages

![Figure 76: Orthogonality of event and resource flow](image)

In figure 76, the general orthogonality scheme between event flow and resource modification is visualized. The event processing builds an event flow by consuming parameters and producing results which can be considered as orthogonal to the resulting state changes of the resources carrying the activity.

In figure 77, the continuous embedding of the three behavioral views providing a seamless behavior modeling is visualized. We can abstract from each implementation using protocol
nets. In a resource allocation net, the request processing actions are refined using service nets. The usual actions and resource pools represent contracts and their usage via service calls in service and resource allocation nets can be interpreted as embedding certain parts of the corresponding protocol net. This *seamless* behavior specification of all these coordination aspects is supported by the embedding of service and protocol nets into the resource allocation net and contracts abstracting from implementation details as visualized in figure 77 and studied in detail in [GGW99c].

The object-oriented analysis and design of systems leads to a decomposition of a system into less complex entities. This decomposition and the *separation of concern* guarantees the *scalability* of our approach. Fortunately in practice, the structuring mechanisms of classes and the hierarchy concept supported by interfaces/contracts and the transparency between textual code and services specified by OCoNs, permits to work with small service nets most of the time.
5.2 The Notation

The different ways OCoNs are embedded into an object-oriented system model are presented using the example considered in detail in [GGW99c]. We recapitulate here the basic elements of the notation and explain their usage in the context of our object-oriented formalism.

![Diagram of OCoN elements](image)

Figure 78: Examples for the elements of an OCoN

An overview about the elements of an OCoN is presented in figure 78. The resource and event pool elements are macros for places in traditional nets. We distinguish between them, to describe the more transient character of parameters, the control flow and temporary events by using event pools as well as the more static resource character of associations and local variables represented by resource pools. Resources are required as the carriers of activities to perform the processing of events during the computation and thus events describe the control flow of a net (see figure 78 (a) and (b)).

A transition named action represents simply an action macro from section 4 with a resource or the active instance itself as carrier of activity. In order to avoid too complex annotations and ensure that the semantic is described visually by the net itself and not determined by the annotations, the usage of the action macros allow to restrict textual annotations to special cases like selection or additional pre- and post-conditions. An action abstraction reduces a behavior to an initial and final step and thus an action does not fire atomically. Instead we assume a logically atomic consume of pre-conditions and synchronous adding of the post-conditions. The signature of an action is visualized as demonstrated by the top action in figure 78(c).

Parameter edges are used to assign the arguments and activation edges determine the object that should be activated. A simple pre-condition arrow stands for consuming one element of the corresponding place and a simple post-condition arrow for producing one element.
Bidirectional activation edges which are only allowed for resources, represent a possibly non-exclusive modifying usage without consume character. The bidirectional parameter edge supplies a copy of the instance and is only allowed for literals or clone-able objects or can be used to specify a pre-condition without any consume character. A pair of consume and produce edges for activation describe the state change for the resource. If an usual remote procedure call is specified the resource is not available in the resulting state until the return arrives. If no activation edge determines a resource for an action, the operation call is directed to the object itself. Optional pre- and post-conditions can be specified in addition to the signature.

A signature is visualized when implementing it with an input bar on the left and output bar on the right side as shown in the service net in figure 78(c). The usage of a corresponding service also reflects these bars as demonstrated in the abstract visual form of an action (figure 78 (c) top). Replications like parallel or sequential for-all loops are specified by edges with double arrows as demonstrated in figure 78(b). Sometimes external activation or sharing should be indicated. We use a shadow-like notation for actions which are controlled externally and double borders for imported resources (see figure 78 (b)). A class contains a resource allocation net for describing the object global coordination, synchronization, its autonomous activity and the resource dependable binding of service calls to service nets or sequential behavior specifications (see figure 75). The set of service nets of a class can specify the behavior of external or internal accessible services. Contracts use OCoNs to describe the external relevant protocol with a protocol net.

5.2.1 Protocol Nets and Contracts

We extend the usual interface notion and combine them with a behavioral description describing the availability of services for an interface (see [Nie95]). These contracts can on a coarse grain level of design be interpreted as architectural connectors [AG97]. Our contract notion is in contrast to [ISO95] restricted to cover only a single interface like, e.g., [Mey97]. We do not consider any functional aspects and specify only the coordination aspect by adding a behavior description.

A protocol net specifies the behavior of a contract and describes those restrictions which are essential to know for an external client when using the offered public services. A contract (contract) is specified using an extension of the UML stereotype interface, where a compartment contains a protocol net. The Order contract shown in figure 79 is build by an interface containing a set of services that allow to initialize the order (setData, addUser), getting the order identifier (getID), test whether an order is overdue (test) and processing the order (process). The protocol net describes the service availability based on abstract external states [Init], [Set], [Accept], [Done] and [Overdue].

The actions in such a protocol net represent available services. An action is equivalent represented in usual P/T nets by a call and return transition, where the initial transition is labeled with the given operator name and the possibly multiple return alternatives are
5.2 The Notation

Figure 79: A contract with protocol net and signatures

named with corresponding return names with indices. For the test operation specification in state [Accept] we get a transition test and two return transitions test1 and test2. In general we use the protocol macros defined in section 3.2.5 which correspond to the call macros reduced to a labeled place transition net by labeling each transition by the corresponding operation name it sends or receives in its action refinement net as defined in section 4.

The well-known procedure call metaphor should ease the understanding of our nets in much the same way as the remote procedure call paradigm has gained much of its benefits from being similar to an ordinary procedure call. The state of the protocol net is represented by a couple of places representing the distinct states of the contract using a state token. Available services for a certain state are represented by actions which lead from this state to another or the same state. Besides the state, we distinguish three different levels of access modes:
5.2.2 Scenarios

When using contracts, for example in a scenario as demonstrated in figure 80, parts of the specified usage protocol can be embedded as needed. This time the actions are not reduced to labeled place transition nets. We instead use the corresponding macro definitions of section 4. In the example scenario, an Order contract is obtained from a Factory resource and the order is initialized (setData), an user is added (addUser), the id is retrieved (getID) and the order is accepted (accept). Thus the actions are service calls, parameters have to be supplied and the termination will deliver a set of results. There has to be a contract represented by a provide port of appropriate type which is providing the activity of an action as well as objects which act as in going parameters or outgoing results for each service call.
5.2 The Notation

In figure 80, the shaded parts show the embedding of the protocol. The resources Factory and Order are used as the principal carrier of activity and the processing itself is triggered using events.

Based on the distinction between resources and objects produced and consumed through the flow of data and control, the metaphor of resources which is crucial in distributed systems can be used to make resource handling explicit. The usage and status of resources can be specified in detail. A single resource may be represented by more than one resource pool if the different pools stand for the same resource but different external states (see the notation introduced in figure 49). The state is annotated with the resource type, e.g., Order[Init].

If the different requirements can not be fulfilled by a decomposition into pools with disjunct state sets the pre-condition edges can be used to specify additional state restrictions for an action. This allows a rather detailed modeling of requirements: for an action to be enabled, the resource is required to be in a specific state, e.g., an Order should be in state [Accept] for activating a getID operation (see figure 79). Also for the polymorphic usage of a set of associations using their common supertype, a resource pool representation can be used (see figure 50).

We distinguish local and imported resources. The former are under exclusive access of the current context whereas the latter are imported from the surrounding context and shared with other behaviors (visualized by double lines; see SecManager resource in figure 81 and 82).

![Diagram of Check permission scenario](image)

Figure 81: Check permission scenario

A second aspect of an order initialization in our example should be a check whether the user has permission to do so. A scenario using a security manager SecManager adds the user if the checkValid request to it succeeds (see figure 81).

5.2.3 Typing

The aspects discussed so far are needed to bring the suitable modeling metaphors for distributed systems into the net language. The aspects discussed in the following are
requirements of a clear integration of object-oriented structural descriptions and nets by integrating the type system provided by the class hierarchy into the nets. The polymorphic type system defined by the class hierarchy provides an isa relation which as usually reflects the position of a class or contract in its class hierarchy. Because an action is identified with a service, the carrier of activity has to be a port/contract of a type and state which provides that service. Moreover, the parameters and results have to be typed objects, too. Hence, pools have associated types and are restricted to markings with typed objects which fit into the pools. Edges with state restrictions or restricted resource pool views might distinguish objects further concerning their external state or combine them in a polymorphic fashion. Simple token are still in use to model the occurrence of events of a basic event type Event to describe pure control flow (see figure 82).

The nets can express calls via the provided actions with signatures. A signature defines – much in the sense of abstract data types [EM85] – simply the types of the parameters and resources connected to an action. It provides the information for static type checking. In and out are visualized by means of ports. We can omit textual annotations for the parameter assignment as usual in high level Petri nets [GL81] using the declaration order to determine the corresponding parameter or result of each port as described in section 4.1.1 (see figure 35 (see figure 35)). The usage of traditional service signatures permits to interface our net formalism with textually provided code of the target language in order to re-use available class libraries, include legacy code or replace a net specification with a hand-coded or automatically generated piece of code. Due to the fact that the entire type system of OCoN depends on the concept of interfaces/contracts and the underlying class hierarchy, it is a simple extension of the well-known mechanisms used in object-oriented languages. This has the additional benefit that an user who is familiar with such a textual language can use the same rules when working with OCoN.

5.2.4 Service Nets

In our example, an OrderCoord object is identified to be responsible for the coordination described in the scenarios of figure 80 and 81. A service net that combines them and specifies how to create and initialize an order according to the specified arguments is presented in figure 82. The instructOrder service might fail if the SecManager detects a security fault. The nondeterministic output of the checkValid service usage in figure 82 is simply a short-hand notation for a behavior which can produce several results like illustrated for example in detail by the instructOrder service.

5.2.5 Resource Allocation Net

In complex cases and for visualizing the initial demanded resources of each service, a description for the instance wide resource allocation and scheduling is needed. A stereotype <<implementation>> with a special resource allocation net compartment is used to build an
5.2 The Notation

Figure 82: A service net to create an order

implementation for the OrderCoord contract as presented in figure 83. The implementation OrderCoordImpl contains a resource allocation net that combines the initial resource demands for all services of its class, provides the point for observing dependencies between services and for specifying design decisions w.r.t. overall resource handling and scheduling within a single instance. Here, resource pools which represent the state and number of the different associated objects and two different kind of actions are used. The state is represented by special state places (self) much like in the protocol nets; special request processing actions, which are specified in section 4.1.4, are drawn with a shadow and are used to specify the initial resource demands for services offered externally. If multiple contract types are provided, the used resource pools are named self.c1 and self.c2 for contract types c1 and c2.

In our example, an inner activity is described in the resource allocation net of figure 83 by several actions in the upper part of the net. All created orders in state [Accept] are tested after a certain delay and transformed into state [Overdue] when their processing was not initiated in time. All orders in state [Accept] are processed by assigning a ProcessUnit
to them if available.

Normally, the protocol net is a less detailed view onto the resource allocation net by omitting those services which are not public as well as abstracting from resources, the states of resources and their influence upon the availability of public accessible services (compare figure 83 and 85).

5.2.6 Class Diagram

We use the abstract requirements for an order processing system to demonstrate the benefits of our approach. Such an order system has to process orders and allow their management.
5.2 The Notation

Security

Processing

Order

OrderCoord

OrderCoordImpl

CoordUser

Ordering

ProcessUnit

SecManager

Figure 84: Class diagram of the order processing system

We have already used several parts of our example design to introduce the elements and explain the semantic for several kind of nets. The overall structure of the design in figure 84 contains a special coordination contract for managing orders and controlling their processing named OrderCoord for each client, which is specified in figure 85. This contract is implemented by OrderCoordImpl that contains a set of Orders represented by a resource orders_ to handle received orders. A security manager SecManager and a resource pool workers_ of ProcessUnits to process the orders are also available for OrderCoordImpl.

The corresponding implementation has already been used as a first example for a resource allocation net in figure 83. There, the coordination of the different services is specified. The relative simple services getOrder or getOverdueOrder which simply retrieve information do not contain any coordination aspects and thus should better left for later implementation with the target language. Hence, they are only found in the textual signature. The instructOrder service is used via a client to add orders. The analysis of the related coordination of an Order contract is described by the scenarios in figure 80 and 81 and its final specification in figure 82. The OrderCoord implementation OrderCoordImpl is an active object that triggers the correct order processing by repeating a test service call to each Order in state [Accept] until the order is either processed by assigning a
ProcessUnit and thus changed to state [Done] or becomes [Overdue]. This is done using a cyclic event flow which is delayed by an internal delay method. A parallel call to all elements of the orders resource pool in state [Accept] is specified using a parallel activation edge. Each single call might transform the order either into state [Overdue] or the old state [Accept]. If an order is in state [Accept], the processing can be initiated by assigning an available ProcessUnit to it via process. For an [Overdue] order, an exceptional handling is needed. Thus we switch the state of the OrderCoord contract using a local action in the resource allocation net which fires when at least one order in state [Overdue] is available (read arc) to signal a client that at least one of its orders has failed.

The client can either use the stornoOrder service specified in figure 86 to simply delete the order or may set a new processing time via the reinstructOrder service described in figure 87. It resets the data and transforms the order to the state [Accept] again. If the client object does not care about failed orders, it can simply ignore the contract state and restrict its usage to instructOrder and getOrder which are available in each state.

5.3 Elements of Object Coordination Nets

The actions introduced in section 4 of the coordination net formalism are the basic entities for the different kinds of nets used for describing a class. The way systems are build is
5.3 Elements of Object Coordination Nets

Figure 86: The stornoOrder service

Figure 87: The reinstructOrder service
restricted by combining nets via port and net creation. Restricting the constructs allowed in each of these net classes ensures their structured usage. So a better structured class description is achieved. We now consider a complete listing of actions, pools and edges used in the several nets to describe their semantic and the intended usage.

5.3.1 Pools

The standard places of a net may contain elements that either represent some kind of resources or logical events modeling the control flow. To distinguish these cases we use the following constructs.

So called resource pools as defined in section 4.2.2 are used to model associations and attributes which are provided by the context of the net. Each such association or attribute is considered to be a possibly empty set of connections or literals. Each such resource pool has an unique port or literal type and name. Restrictions like a [state] are annotated using square brackets. The multi-set view on resources allows to interpret a single entity simply as a special case. If the pool is an imported one, this is visualized using a double border. In protocol nets resource pools are also used to specify the different states of a contract.

When an operation available in every state is requested, we do not need the protocol state as pre-condition. Thus, we can also process a call during a running exclusive protocol usage. We omit for these cases the pre-condition edge to the resource pool and instead use an edge with textual annotation. If the net or class behavior does not ensure the existence of a suitable resource for each possible occurrence a non-enabling edge has to be used and an exception might occur.

Resource pools for class wide clone-able constants can be omitted and we simply add the constant name to an edge which is not connected to a pool or resource graphically.

For holding temporary events, so called event pools as defined in section 4.2.1 are used. Like the resource pools they handle a number of connections or literals. The difference to a resource pool is the restriction that events must always be consumed and can neither be used as additional condition (read arc), for non enabling edges, nor for set oriented access. So, events are used only to describe the progress and control flow of a net.
5.3 Elements of Object Coordination Nets

5.3.2 Call Actions

Transitions in Petri nets extended with send, receive, port and net creation are much too flexible and low level to provide an easy to use design notation. So we restrict their usage to special cases and provide a set of actions to describe the usual cases of interaction via interfaces encapsulating all the details. Transitions as atomic steps in traditional Petri nets are replaced by actions, which in contrast to transitions do not fire atomically.

Each call action represents a service call and thus can be seen as a combination of an atomic call transition and a return transition as demonstrated in section 4.1.1, where the call action is defined. In protocol nets only the resulting labeling is used to specify the protocol.

When a specific service net should be called with the object context, we use the direct net invocation action defined in section 4.1.3. This allows to call internal methods and avoids to synchronize the request with other external requests in the resource allocation net. Thus the internal call does not have to be scheduled together with external requests. Also sub-behaviors can be specified locally only accessible for a single service net to provide more abstract partial behavior and improve the scalability of behavior descriptions with service nets.

In both cases, we distinguish further between remote procedure calls, remote procedure calls with asynchronous reply and so called one way calls. A remote procedure call needs as pre-condition an appropriate protocol state and locks the state part of a contract until the return occurs. In contrast, the remote procedure call with asynchronous reply splits the contract and instead releases the state part directly after the call has occurred. The reply is processed using an orthogonal running throwaway contract. The one way call also only blocks the state part until the call is initiated.

The opposite of an one way call which is indeed a send operation is a simple receive operation. An explicit management of single receives would conflict with the design restriction that all external requests are handled by the resource allocation net and thus would complicate the overall handling of contracts to a great extent. We thus do not support them on this level and instead use contract state changes. Such a corresponding state change is used to abstract from possible receives on the contract usage side (see section 3.2.6).

We restrict OCoNs to only use these combined kinds of transition to represent a "method call", which is usually the basic unit of interaction. We can thus avoid a semantic gap, which occurs when message oriented formalisms are directly used to describe call oriented systems.
5.3.3 Invocation Actions

To model the creation of new nets as a reaction to service calls in an arbitrary fashion using, for example, special kinds of strategies to increase efficiency, is not necessary at the design level. Instead, such details are ignored and only the question, which net should be used for a request, is of concern. It is remarkable that this way of using Petri nets for handling requests or internal tasks induces no artificial limit to concurrency as a process oriented formalism might do. We restrict the net creation for request processing to appear only in the form of a special action in the resource allocation net.

To process incoming requests in the resource allocation net and forward them to a dynamically created service net, the call forward action from section 4.1.4 can be used. If an operation has an unique service net the corresponding annotation $N$ can be omitted. The annotated edges are only necessary to add resources of the class environment, that are needed exclusive, to the service call. So the initial resource consumption of an invocation can be specified and for example different service nets which are suitable for different situations can be activated depending on the resource availability.

Versions for a remote procedure call, a remote procedure call with asynchronous reply and one way call exist. The version for the one way call is like the one for the asynchronous reply with a create and terminate part to release the exclusive needed resources, but no result is send after termination.

The service net signature is the combination of the operation signature plus the exclusive needed resources. Thus a correct usage ensures that all exclusive needed resources or additional parameters must be provided to the call forward action in the resource allocation net. If needed resources are determined dynamically, they can not be specified here of course. By importing all resources used in the corresponding service net via imported resource pools automatically, these resources are available, too, which results in the expected sharing of instance-wide accessible resources.

So, send, receive and net creation are not accessible directly, instead we have to use the introduced actions.

5.3.4 Internal Actions

For local transitions only port creation and moving, copying and deleting data is allowed.
5.3 Elements of Object Coordination Nets

Only when events, resources and simple tokens should be moved or literals, shared ports or records of both should be copied, it is allowed to additionally use local transitions. We use them also to describe internal protocol steps, where the activity edge determines the source and destination state of the internal step.

The local transitions are restricted to local effects and receive, send or create annotations are not permitted. Also no guard terms are allowed to exclude complex or even oracle like local behavior of a net. Thus a simple and feasible local step semantic can be achieved.

Their effects are guaranteed to be atomic, which indeed only means, that no temporary state occurring during their processing is visible for any other action. The visual representation that is not a combination of two transitions but like a single one, should indicate the atomic nature.

The progress guaranteeing version is simply a white square and a quiescent variant can also be used to model non-determinism.

Both are only allowed to appear in service nets and resource allocation nets, because a protocol net is describing the possible usage from an external point of view.

5.3.5 External Actions

When internal effects of a resource allocation net or induced behavior caused by interaction of multiple clients with one contract has to be described, corresponding external transitions can be used. They allow to model the external visible abstract state of an interface in a protocol net. We distinguish them from the local transitions by using an additional shadow, because in situations where the protocol should be embedded, a mix might occur and thus a different graphical representation excludes any ambiguous situations.

The external transitions are drawn with a shadow to implicate that they are triggered externally. This is done in conformance with the shadow for transitions with receive expressions in coordination nets and the call forward action. This kind of behavior occurs in protocol nets when spontaneous state changes caused by the contract provider are specified. An optional embedding of these transitions when a resource with the specific protocol net type is used, leads to a visual representation that includes these spontaneous state changes. Thus in a net using a resource depending on its protocol, the whole behavior specified by client interaction and spontaneous protocol steps itself is visible and the coordination becomes more obvious.
5.3.6 General Treatment of Actions

For all actions some special configuration and usage rules can be applied. This includes graphical foldings for signatures and what kind of pre- and post-conditions are allowed.

An action may have multiple return alternatives visualized by splitting the return transition accordingly. The resulting areas are called return parts of the action. This folding introduced already for the call action allows to model different reactions of a serving operation still as a single action rather than splitting them into several transition as usually done in Petri nets.

Each of the resulting return parts and the call transition has a type described by some kind of signature. The call transition list describes the number of input parameters, the corresponding service call needs. For each alternative result, the list of each return part describes which results are provided. This information is visualized using parameter ports as defined in section 4.1.1. The input parameters are visualized using white circles and the output parameters by black ones. The top to bottom order is kept for 0 or 180 degree rotations and transformed to the textual declaration order from left to right if rotated by 90 or 270 degree.

The fact that an action is equivalent concerning its local behavior to two atomic transitions, results in the theoretical ability to add post-conditions to the service call and pre-condition to the service return as described in section 4. The latter one is in conflict with the expected procedure call semantic and is thus not allowed. But allowing post-conditions to a call is sometimes very useful. For example, when a service call atomically changes the interface state. This can only be modeled when using service call post-conditions. Another useful scenario is when two alternative calls are specified and right after one of them is chosen, additional work should be started. If no post-conditions for the initial part of a call are allowed, the needed additional work is delayed after the chosen call really terminates.

A so called one way operation of an interface is restricted to always avoid any return synchronization. It obviously could not have any return results or exceptions caused by the serving object. It is visualized by omitting the return transition part and thus return post-conditions are not possible.
5.3 Elements of Object Coordination Nets

5.3.7 Edges for Parameters, Activation and Additional Purposes

Different kinds of edges are used for parameter and port assignment. Also additional edges can be used with the flexible high level Petri net transition semantic based on assignments in a restricted fashion. The high level Petri net standard uses a flexible mechanism to describe the involved elements of a transition that allows for example to name sub-elements of a record \( v \in [\text{int} \, \text{num}; \text{int} \, \text{val}] \) by using an expression \(<x, y>\) with variables \(x\) and \(y\). This is shorter and more flexible than using a more object-oriented style for records like \(v.num\) and \(v.val\). This conflict of the high level Petri net standard with the target notations like CORBA IDL or DCOM IDL can be neglected and we assume simply that both kinds of styles are possible and can translated into each other in an obvious way.

In analogy to object-oriented languages, we distinguish parameter and activation edges for call actions. An activation edge is drawn with a white arrow head in contrast to parameter edges using a black arrow head. Additional edges are drawn like parameter edges, but can be distinguished, because they end or begin at the action border. The always unique activation edge determines which port will be used for communication. Parameter edges add the needed parameters for this call.

For a call forward action, the unique activation edge specifies the port whose requests are processed. Parameter edges are not allowed, because the parameters are already provided by the caller. Additional edges are used to specify the exclusive imported environment.

When additional pre-conditions are needed, they can simply be attached to the action itself using an edge to the border of the action. They can express a pre-condition event that is consumed but not further considered. By naming the edge, the contents is made accessible for further usage: For a resource, a read arc can be used to either obtain a copy of a clone-able object by naming the edge or simply specifying the existence as pre-condition when omitting any naming.

Additional post-conditions can analogously be specified by an edge from the return part border to any kind of pool. If they carry no additional information, they must be of the basic event type Event. Otherwise a corresponding variable evaluated in the action context determines the resulting object or literal.

This way additional edges are used to extend the call context of the calling net. Such extensions bind a value to the specified variable which can then be used in selection expressions or post-condition edges.
The activation edge explicitly demands, that already a protocol state is reached that ensures immediately acceptance of that request or caller and callee may synchronize to ensure the immediate processing. The usage of synchronous or asynchronous send is left open for further refinement.

For a parameter edge, an edge without crossbar simply denotes a precondition.

For both kinds of edges holds that if a crossbar is drawn we have a non enabling edge and the described element is not determined as an enabling pre-condition. Instead, when all enabling pre-conditions are given the non enabling edges are evaluated and consumed when the action is already started. If the demanded elements are not available, an exception occurs.

The resulting behavior contains some sequential ordering considering the edge evaluation, but we still can achieve the action abstraction for non enabling edges, because of the non blocking semantic for the non-enabling edge consume or read (see subsection 4.5.2 for further details).

Variables bound as a parameter or by an additional edge can be used in so called selection expressions. To exclude complex guard expressions, an edge can either be named or use a selection expression which is restricted to non-enabling edges.

Set-oriented access has always to occur with non-enabling edges and thus replication is only supported with non-enabling edges. For a detailed description of the related semantic see subsection 4.3. Informally, we can describe the replication and selection with non-enabling edges as an iterative process that visits all possible candidates during the action is running and terminates either when one suitable candidate has been found and visited (selection) or all have been visited (set selection). If no selection criteria are given for a set-oriented selection all resources of the corresponding resource that already exists when the action is started have to be visited. Replication can also be used in combination with a given set by evaluating corresponding selections for each element in parallel or sequential. All usual pre-conditions have to be provided by cloning them in the parallel case or by pipe the results of stage \( i \) as input parameter into stage \( i + 1 \).
5.4 Conclusion

To offer the ability to catch exceptions thrown by an action, the called object/contract or the local action pre-condition evaluation might throw them. Additionally when selection expressions are used, errors like "not expected empty selection" might also be handled by this mechanism. A special exception edge is used to describe the target pool, the thrown exception object is put in. For multiple exception edges, the target event pool type specifies which kinds of exceptions are caught by each edge (see section 4.5.3 for further details).

Other tasks like throwing an exception or create a durable net are also specified using the actions specified in section 4.

5.4 Conclusion

The presented OCoN project covers several important aspects of software engineering for distributed systems. For the structural modeling of systems the concept of object-orientation more concrete the UML is applied. In this report the embedding of OCoN nets to describe object-oriented systems design with contracts and object behavior is presented. A detailed discussion of the UML behavior notations and their disadvantages in comparison with the OCoN approach are discussed in [GGW99a]. Distributed system design adds its share to system complexity. The aspect of parallel processing or concurrency and their impact on the design and further work on OCoNs with emphasis on the contract design can be found in [GGW99b, Gie99]. In contrast to the high level Petri net standard and the suggested extensions for Petri nets concerning object-orientation the described OCoN net dialect has been designed with emphasis on it usability as a visual languages. The mentioned seamless visual embedding of the approach and its advantages in the context of visual languages are discussed in [GGW99d, GGW99c, WGG99]. The OCoN approach has evolved form its beginning [WGG97] to a fully developed visual behavior specification mechanism that covers all aspects of object behavior [GGW98]. It has been evaluated in several system design projects and is subject to continuous reviews with regard to usability.
A  High Level Petri Net Standard Conformance

To achieve level 2 conformance with the ISO high level standard draft [Com97], a mapping from coordination nets to a high level Petri net graphs and a mapping to high level Petri nets is described.

A.1 Mapping to High-Level Petri Nets

Starting with a coordination net system \( CNS = (\mathcal{N}, \mathcal{S}, \mathcal{T}, \mathcal{C}, \Sigma^c, \mathcal{M}_0, \mathcal{P}_0, \mathcal{R}_0) \) we have to define a corresponding behavioral equivalent high level Petri net

\[
HLPN = (S_{HLPN}, T_{HLPN}, C_{HLPN}, C_{HLPN}, \text{Pre}_{HLPN}, \text{Post}_{HLPN}, M_0)
\]

The set of places \( S_{HLPN} \) is defined by extend the global place set \( \mathcal{S} \) with two special places for the queues and the reverse.

\[
S_{HLPN} := \mathcal{S} \cup \{s_{\text{QUEUE}}, s_{\text{REVERSE}}\}
\]

The typing for all places \( s \in S_{HLPN} \) can be defined based on the global typing function \( C \) of the coordination net system

\[
C_{HLPN}^1(s) = \begin{cases} 
\mathbf{Id} \times C(s) & s \in \mathcal{S} \\
\text{QUEUE} & s = s_{\text{QUEUE}} \\
\text{REVERSE} & s = s_{\text{REVERSE}} 
\end{cases}
\]

The set of transitions \( T_{HLPN} \) is defined based on the global transition set \( \mathcal{T} \) by add every non synchronizing transition and all matching pair of synchronizing transitions.

\[
T_{HLPN} := \{t \in \mathcal{T} | \exists (id, t, \alpha) \in \text{TRANS}_\text{net}\} \\
\cup \{(t, t') \in \mathcal{T}^2 | \exists ((id, t, \alpha), (id', t', \alpha')) \in \text{TRANS}_\text{sync}\}
\]

The net instances are encoded in section 3.3 by adding the net instance number to every local place marking and ensure via a special \( id \) element that every transition \( (id, t, \alpha) \) only use local places of the same net instance or shared ones of a parent instance. This can be embedded into a high level Petri net by define the modes (types) for a transitions \( t \) as \( (id, \alpha) \) and for a transitions pair \( (t, t') \) as \( ((id, \alpha), (id', \alpha')) \).

\[
C_{HLPN}(x) = \begin{cases} 
C_{HLPN}^1 & x \in S_{HLPN} \\
\{((id, \alpha), (id', \alpha')) | ((id, x, \alpha), (id', x, \alpha')) \in \text{TRANS}_\text{net}\} & x \in \mathcal{T} \\
\{((id, \alpha), (id', \alpha')) | ((id, x, \alpha), (id', t, \alpha')) \in \text{TRANS}_\text{sync}\} & x = (t, t') \in \mathcal{T}^2
\end{cases}
\]

The finite set of types \( C_{HLPN} \) can now be build using the image of \( C_{HLPN} \) for \( S_{HLPN} \) and \( T_{HLPN} \)

\[
C_{HLPN} := \{R | \exists s \in S_{HLPN} \ R = C_{HLPN}(s) \lor \exists t \in T_{HLPN} \ R = C_{HLPN}(t)\}.
\]
A.2 Mapping to High-Level Petri Net Graph

So we get the set of transitions

\[ \text{TRANS}_{HLPN} := \{(t, m)|t \in T_{HLPN}, m \in C_{HLPN}(t)\} \]

and the set of places

\[ \text{PLACE}_{HLPN} := \{(s, g)|s \in S_{HLPN}, m \in C_{HLPN}(s)\}. \]

An initial Marking \( M_0 \) can be built from the initial marking \( M_0, P_0 \) and \( R_0 \) by map them on the corresponding places \((id, s) \in S_{HLPN}\).

\[ M_0(s) := \begin{cases} M_0(s) \in \mu C_{HLPN}(s) & s \in S \\ P_0 \in \mu \text{QUEUE} & s = s_{\text{QUEUE}} \\ R_0 \in \mu \text{REVERSE} & s = s_{\text{REVERSE}} \end{cases} \]

The resulting pre- and post-conditions can be used analogous to define suitable pre and post mappings \( \text{Pre}_{HLPN}, \text{Post}_{HLPN} : \text{TRANS}_{HLPN} \to \mu \text{PLACE}_{HLPN} \) by the equations

\[ \text{Pre}_{HLPN}(t)(s) := \begin{cases} \text{Pre}_M^t(s) & s \in S \\ \text{Pre}_P^t & s = s_{\text{QUEUE}} \\ \text{Pre}_R^t & s = s_{\text{REVERSE}} \end{cases} \]

and

\[ \text{Post}_{HLPN}(t)(s) := \begin{cases} \text{Post}_M^t(s) & s \in S \\ \text{Post}_P^t & s = s_{\text{QUEUE}} \\ 0 & s = s_{\text{REVERSE}} \end{cases} \]

which are extended to the multi-set case using the linear extension

\[ \text{Pre}_{HLPN}(T_\mu) := \sum_{t \in T_\mu} T_\mu(t) \text{Pre}(t) \quad \text{and} \quad \text{Post}_{HLPN}(T_\mu) := \sum_{t \in T_\mu} T_\mu(t) \text{Post}(t). \]

Thus we finally have the demanded behavioral equivalent *high level Petri net* for a given *coordination net system*

\[ HLPN = (S_{HLPN}, T_{HLPN}, C_{HLPN}, C_{HLPN}, \text{Pre}_{HLPN}, \text{Post}_{HLPN}, M_0). \]

A.2 Mapping to High-Level Petri Net Graph

To achieve the level 2 conformance a mapping to the *high level Petri net graph* syntax is presented. For a given *coordination net system* \( CNS = (N, S, T, C, \Sigma, M_0, P_0, R_0) \) we have to define a corresponding *high level Petri net graph* \( HLPNG = (NG, \Sigma, C, AN, M_0) \).

Without any restriction we can assume, that for every distinct pair of elements of the set of *coordination net graph* \( N \neq N' \in N \) holds \( T^N \cap T^{N'} = \emptyset \). The places and transitions and their typing can be determined like for *high level Petri nets*. But additional edges and a
modified set of annotations have to be added. For the omitted reverse variables and queue handling additional edges and corresponding annotations are needed to achieve a mapping to high level Petri net graphs. The flow relation for the single transitions is accordingly described by

\[ F_{HLPN}^T := \bigcup_{N \in \mathcal{N}} \{(s, t) \in F^N | t \in \text{TRANS}_{HLPN}\} \]

\[ \cup \{(s_{\text{REVERSE}}, t) | \exists t \in \mathcal{T} \text{ free}(TC(t)) \cap V_{\text{REVERSE}} \neq \emptyset \} \]

\[ \cup \{(s_{\text{QUEUE}}, t) | \exists t \in \mathcal{T} (TC(t) \cap \text{TERM}(\Omega \cup V)_{\text{comm}} \neq \emptyset) \} \]

For the transitions build by a pair \((t, t') \in \text{TRANS}_{HLPN}\) we have to consider the following edges

\[ F_{HLPN}^{T^2} := \bigcup_{N \in \mathcal{N}} \{(s, t) \in F^N | \exists t' \in \mathcal{T} ((t, t') \in \text{TRANS}_{HLPN} \vee (t', t) \in \text{TRANS}_{HLPN})\} \]

\[ \cup \{(s_{\text{REVERSE}}, (t, t')) | \exists (t, t') \in \text{TRANS}_{HLPN} (\text{free}(TC(t)) \cup \text{free}(TC(t'))) \cap V_{\text{REVERSE}} \neq \emptyset \} \]

\[ \cup \{(s_{\text{QUEUE}}, (t, t')) | \exists (t, t') \in \text{TRANS}_{HLPN} \} \]

And thus finally we get the flow relation \(F_{HLPN} = F_{HLPN}^T \cup F_{HLPN}^{T^2}\).

The annotations for the edges and transition \(t \in \text{TRANS}_{HLPN} \cap \mathcal{T}\) are then defined by

\[ \forall t \in \text{TRANS}_{HLPN} \cap T^N : \text{TC}_{HLPN}(t) := TC^N(t). \]

For all transitions \(t \in \text{TRANS}_{HLPN} \cap T^N\) and \((s, t) \in F^N\) we have to extend each expression \(x\) to \((id, x)\) to add the needed net index. This is done using a function \(\text{extend} : \text{TERM}(\Omega \cup V)_{id} \times \text{BTERM}(\Omega \cup V)_r \rightarrow \text{BTERM}(\Omega \cup V)_{id \times r}\) which extends every multi-set of expressions and add a net index expression \(\text{idLocate}(id, s)\) to express that the consumed token \((s, id', v)\) is virtually in the real place \((s, id)\) for \(id' = \text{idLocate}(id, s)\).

\[ A((s, t)) := \text{extend}(\text{idLocate}(id, s), A^N(s, t)) \]

The same is done for every post-condition edges \((t, s) \in F^N\)

\[ A((t, s)) := \text{extend}(\text{idLocate}(id, s), A^N(t, s)). \]

For annotations of combined transitions \((t, t')\) the simple combination of the annotations of \(t\) and \(t'\) is not possible because shared variables may cause unwanted semantical influences. By assuming a variable renaming function \(\text{rename}_{(t, t')}\) for terms and variables we can avoid this problem.

\[ \forall (t, t') \in \text{TRANS}_{HLPN} \cap T^N \times T^N' : \exists (s, t) \in F^N \]

\[ A((s, (t, t'))) := \text{extend}(\text{idLocate}(id, s), A^N(s, t)) \]

and \(\forall (t, t') \in \text{TRANS}_{HLPN} \cap T^N \times T^N' : \exists (s, t') \in F^{N'} \)

\[ A((s, (t, t'))) := \text{extend}(\text{idLocate}(id', s), \text{rename}_{(t, t')}(A^N(s, t'))) \]
A.2 Mapping to High-Level Petri Net Graph

For the post-conditions we get $\forall (t, t') \in \text{TRANS}_{HLPN} \cap T^N \times T'^N \; \exists (t, s) \in F^N$

$$A(((t, t'), s)) := \text{extend}(\text{idLocate}(id, s), A^N(t, s))$$

and $\forall (t, t') \in \text{TRANS}_{HLPN} \cap T^N \times T'^N \; \exists (t', s) \in F'^N$

$$A((s, (t, t'))) := \text{extend}(\text{idLocate}(id', s), \text{rename}(t,t')(A^N(t', s))))$$

For a synchronous pair the fresh message variable $m_{s}$ of the send expression and the receive $m'_{s}$ of the receive expression are set equal using a boolean term built by the following function combine:

$$\text{TERM}(\Omega \cup V)_{ssnd} \times \text{TERM}(\Omega \cup V)_{rcv} \rightarrow \text{TERM}(\Omega \cup V)_{\text{Bool}}.$$

$$\forall (t, t') \in \text{TRANS}_{HLPN} \cap T^N \times T'^N$$

$$\text{TC}_{HLPN}(t, t') := (\text{TC}^N(t) + \text{rename}(t,t')(\text{TC}^N(t)))$$

$$+ \text{combine}(\text{TC}^N(t) \cap \text{TERM}(\Omega \cup V)_{ssnd}, \text{TC}^N(t') \cap \text{TERM}(\Omega \cup V)_{rcv}).$$

For all message sending and receiving the corresponding edge annotations for a single transition $t \in \text{TRANS}_{HLPN} \cap T$ must be

$$\forall (\text{sQUEUE}, t) \in S_{HLPN} \; t \in T^N$$

$$A(\text{sQUEUE}, t) := \sum_{\eta = \zeta \triangleright \langle m \rangle / \phi, w \in \text{valTC}^N(t)} (\phi, m; w) + \sum_{\eta = \zeta \triangleright \langle m \rangle / \phi, w \in \text{valTC}^N(t)} (\phi, w)$$

$$\forall (t, \text{sQUEUE}) \in S_{HLPN} \; t \in T^N$$

$$A(t, \text{sQUEUE}) := \sum_{\eta = \zeta \triangleright \langle m \rangle / \phi, w \in \text{valTC}^N(t)} (\phi, w) + \sum_{\eta = \zeta \triangleright \langle m \rangle / \phi, w \in \text{valTC}^N(t)} (\phi, w; m)$$

For a pair $(t, t') \in \text{TRANS}_{HLPN} \cap T^2$ we get

$$\forall (\text{sQUEUE}, (t, t')) \in S_{HLPN} \; t \in T^N \; t' \in T'^N$$

$$A(\text{sQUEUE}, (t, t')) := \sum_{\varphi = \psi \triangleright \langle m \rangle / \text{valTC}^N(t)} (\phi, \epsilon) + \sum_{\varphi = \psi \triangleright \langle m \rangle / \text{valTC}^N(t')} (\phi, \epsilon)$$

$$\forall ((t, t'), \text{sQUEUE}) \in S_{HLPN} \; t \in T^N \; t' \in T'^N$$

$$A((t, t'), \text{sQUEUE}) := \sum_{\varphi = \psi \triangleright \langle m \rangle / \text{valTC}^N(t)} (\varphi, \epsilon) + \sum_{\varphi = \psi \triangleright \langle m \rangle / \text{valTC}^N(t')} (\varphi, \epsilon)$$

For all variables of the reserve the corresponding edge annotations must be

$$\forall (\text{sREVERSE}, t) \in S_{HLPN} \; t \in T^N$$

$$A((\text{sREVERSE}, t)) := \text{free}(\text{TC}^N(t)) \cap V^{\text{REVERSE}}$$

$$\forall (\text{sREVERSE}, (t, t')) \in S_{HLPN} \; t \in T^N, t' \in T'^N$$

$$A((\text{sREVERSE}, (t, t'))) := \text{free}(\text{TC}^N(t)) + \text{rename}(t,t')(\text{free}(\text{TC}^N(t'))) \cap V^{\text{REVERSE}}$$

Thus finally all additional elements of a coordination net system are translated into an equivalent high level Petri net graph as defined in the high level Petri net standard.
References


REFERENCES


[Che91] NewsletterInfo: 40.


REFERENCES


REFERENCES


[ISO95] ISO/IEC. Open Distributed Processing Reference Model - parts 1,2,3,4, 1995. ISO 10746-1,2,3,4 or ITU-T X.901,2,3,4.


REFERENCES


[MM97] Christoph Maier and Daniel Moldt. Object Colored Petri Nets - a Formal Technique for Object Oriented Modelling. Workshop PNSE’97, Petri Nets
in System Engineering, Modelling, Verification, and Validation, Hamburg, Germany, September 1997.


REFERENCES


List of Figures

1  A Petri net and a corresponding place transition net .................. 9
2  Reachability graph ................................................................... 11
3  A run for the place transition net of figure 1 ......................... 13
4  A net with fair and quiescent transitions ............................... 15
5  A high level Petri net example ............................................. 19
6  A second high level Petri net example ................................... 19
7  Requirements for modeling an object ..................................... 23
8  Basic structure of a coordination net system ........................... 24
9  A coordination net graph example ........................................ 25
10 An example for different ways nets may interact .................... 26
11 An example for an asynchronous communication .................... 27
12 Syntactically restricted message formats on a port .................. 28
13 The client server connection for a remote procedure call .......... 29
14 The protocol net related to a remote procedure call ............... 30
15 The usage side of a remote procedure call ............................. 30
16 General protocol for multiple supported remote procedure calls 31
17 A realistic protocol for a file handle ..................................... 31
18 Connection situation during a running request ....................... 32
19 Protocols of a remote procedure call with asynchronous reply 32
20 A call and return with temporary return port ......................... 33
21 A macro for a call abstraction ............................................. 34
22 A macro for a call with asynchronous reply .......................... 34
23 A macro for an one way call .............................................. 34
24 Macros for different internal steps ...................................... 34
25 Macro for a durable or transient state .................................. 35
26 A protocol with internal steps ............................................. 35
27 The file protocol specified using the macros ......................... 36
28 Creation tree and related location of imported places ............. 48
29 An example run for a coordination net system ...................... 57
30 A second example run for a coordination net system ............. 57
31 The two step action abstraction ........................................ 60
32 Behavior relation for an action ......................................... 61
33 Embedding of an action .................................................. 61
34 Embedding of an action with multiple output alternatives .......... 62
35 Visual representation of signatures ..................................... 62
36 Folding for call and return with temporary return port ........... 62
37 Alternative return vectors can be fold .................................. 63
38 A call action for an invocation on a port with alternative return vectors 63
<table>
<thead>
<tr>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>82</td>
<td>A service net to create an order</td>
</tr>
<tr>
<td>83</td>
<td>Resource allocation net (OrderCoordImpl)</td>
</tr>
<tr>
<td>84</td>
<td>Class diagram of the order processing system</td>
</tr>
<tr>
<td>85</td>
<td>The external view on the order system</td>
</tr>
<tr>
<td>86</td>
<td>The stornoOrder service</td>
</tr>
<tr>
<td>87</td>
<td>The reinstructOrder service</td>
</tr>
</tbody>
</table>
List of Tables

1  Processing of `addDownTime` sequential replication  ....................... 77
Index

Symbols
Σ-inscription, 13, 56

A
abstract action net, 60, 61
abstraction, 27
ACID properties, 71
action, 59-62, 89, 101
call, 62
call forward, 65
call with asynchronous reply, 63
net creation and invocation, 64
one way call, 64
action refinement net, 60
activation
tedge, 105
activation edge, 33, 80, 105
acyclic, 13
additional edge, 105
Allen, , 5
alphabet, 10, 37
architectural connectors, 5
assignment, 18, 19, 50, 54
asynchronous, 22, 24, 35, 38, 106
asynchronous reply
remote procedure call, 32
asynchronous send, 26, 43, 50
atomic, 67, 103
autonomy, 27

B
Battiston, E., 20
behavior, 27
Bernardinello, Luca, 15
Best, Eike, 21
binding, 46
block, 59
Brauer, Wilfried, 59
Buchs, Didies, 20

Busi, Nadia, 22

C
call, 62, 63
call action, 62, 64, 101, 105
call forward action, 65, 66, 69, 102, 103, 105
call transition, 101
callee, 29, 32, 106
caller, 29, 32, 62, 105, 106
can not process message, 28
carrier of activity, 89
cast, 79
cast expression, 79
characteristic function, 7
characteristics, 27
Chehaibar, Chassan, 59
Christensen, Soren, 59
client, 28
client/server, 20, 29
colored Petri net, 15, 59
communicating net, 20
concurrency, 5
concurrent enabling, 54
concurrent occurrence, 54
condition event net, 13
consistency, 40
consistent, 40, 41, 49
constant, 100
constructor, 78
consume, 66
contract, 29, 32, 90
control flow, 100
cooperating net, 20
COOPN, 20
coordination, 5, 103
coordination net, 22, 23, 36, 43, 59, 83, 103, 108
definition, 36
introduction, 23
semantic, 49
coordination net graph, 23, 45, 48, 49, 60, 109
coordination net system, 23, 36, 48, 49, 56, 108, 109
coordination net systems, 14
copy
   literal, 103
   shared port, 103
CORBA, 21, 24, 27, 28, 34, 37
Crnogorac, Lobel, 83
cross product, 8
CSP, 10

D
DCE, 28
DCOM, 21, 24, 27, 28, 37
De Cindio, Fiorella, 15
deadlock, 28, 40
definition
   coordination net, 36
   word, 8
direct net invocation action, 64, 101
distributed object system, 5
distributed software system, 5
distributed system
   specification, 5
divergence, 10
divergent, 10
durable, 29, 35, 66
durable state, 41
dynamic scoping, 48

E
edge
   activation, 105
   parameter, 105
edge weight, 8, 9
elementary object net, 20
enabled, 9, 12, 52, 54–56
enabling, 49, 106
Engberg, U., 22
Engelfriet, Joost, 22
event
   move, 103
event pool, 66, 84, 100
exception, 31, 100, 107
exception edge, 107
exclusive, 33, 38–41, 44, 86, 102
exclusive connection, 40
exclusive state step relation, 40
expression
   cast, 79
ExSpect, 20
external transition, 103

F
failure, 29
failure set, 11
failure trace model, 10
failure trace semantic, 11
failures, 11
fair, 15, 58
fairness, 14, 15, 58
FIFO, 20, 23, 38
FIFO Petri net, 21
finite, 8
finite Petri net, 8, 17, 45
finite state machine, 11
Fleischhack, Hans, 21
flow relation, 8
fresh, 43
function, 7
   injective, 7
   partial, 7
   surjective, 7
   total, 7

G
Garlan, David, 5
Genrich, H. J., 15
Gorrieri, Roberto, 22
Grahlmann, Bernd, 21

H
Hansen, N. D., 59
Hewitt, C., 22
indexed colored Petri net, 19
high level Petri net, 15, 18, 19, 36, 59, 108, 109
ISO standard, 15
high level Petri net graph, 17, 18, 49, 108–110
Hoare, C.A.R, 10
Honda, Kohei, 22
Huber, Peter, 19

I
import place, 46
imported, 69
imported resource pool, 102
inheritance, 27, 28
inheritance anomalies, 83
inhibitor arcs, 59
initial labeled steps, 10
initial marking, 46
initial port marking, 49
initial reverse universe, 49
initial system marking, 49
injective function, 7
inner activity, 95
input, 78
input parameters, 104
interaction, 27
interface, 27–29, 37
interface typing, 83
interpretation, 17
invocation, 65
irreflexive, 13
ISO
high level Petri net, 15

J
Java, 21, 83
Jensen, Kurt, 15, 19

K
Kiehn, A., 21
Kindler, Ekkart, 15

L
labeled transition system, 11
labeling function, 10
Lakos, Charles, 20
Lautenbach, Kurt, 15
Lea, Doug, 87
Lilien, Johan, 21
linear extension, 12, 16, 54, 55, 109
literal, 67
copy, 103
move, 103
literal typing, 27
literals, 27
live lock, 28
local, 66, 69
local place, 46
local transition, 102, 103
location, 48
lock, 66
manual, 69
LOOPN, 20
LOOPN+, 20

M
M-Nets, 21
many-sorted algebra, 17
marking, 8, 9, 49
marshaling function, 24, 38
Memmi, G., 21
message not understood, 28
method, 65
middleware, 37
Milner, Robin, 22
Mobile Nets, 22
mode, 19
Moldt, Daniel, 20
Montanari, Ugo, 71
move
event, 103
literal, 103
resource, 103
multi-set, 7
multi-set cross product extension, 8, 18
multi-set inclusion, 7
multi-set subtraction, 7
multi-set sum, 7
Murata, T., 59

N
Najm, Elie, 22
net, 14
net creation and invocation action, 64
never processed messages, 28
new action, 82
Nielsen, , 22
non enabling, 106
non enabling edges, 80
non-determinism, 103
non-enabling, 100, 106
not decidable, 47

O
object, 24
object based, 20, 27
object coordination net, 6, 23
   definition, 83
   introduction, 83
object life cycle, 27
object-orientation, 5, 107
object-oriented, 27, 42
object-oriented design, 5
object-oriented languages, 105
objects, 5
OBJSA net, 20
obligation, 29
occur, 9, 12, 14, 54
occurrence, 49
occurrence net, 7, 12
occurs, 56
OCoN
   definition, 83
   introduction, 83
OCP net, 20
one way call, 34, 65, 101, 102
one way call action, 64
OP-Nets, 20
operation, 100
optional behavior, 29
OQL, 71
oracle, 103
oracle free, 47
output, 78
output parameters, 104

P
parallel replication, 71
parameter
   edge, 105
parameter edge, 80, 106
partial function, 7
partial order, 12
passive, 27
Pelz, Elisabeth, 21
PEP, 21
Petri net, 7, 8, 101
Petri net semantic, 18, 22
place, 8
   local, 46
place fusion, 19
place import, 46
place substitution, 19
place transition net, 7, 8, 11, 13, 16
PN, 83
PNtalk, 20
polymorphic, 68
pool
   event, 100
   resource, 66, 100
port
   typed, 27
port passing calculi, 22
port typing, 27
post-condition, 9, 12
power-set, 7
pre-condition, 9, 12, 106
predicate transition net, 15, 22
procedure call semantic, 104
produce, 66
progress, 14, 15, 29, 34, 100, 103
progress guarantee, 14
progress transition, 29
PROT, 20
protocol, 29, 32, 37
protocol net, 83, 84, 86, 90, 92, 103
protocol state, 37, 38
provide port, 25, 37, 92
provider, 28
proxy, 87
Puntigam, Franz, 22

Q
quiescent, 15, 29, 34, 58–60, 62, 103
quiescent transition, 29

R
Rambags, P. M. P., 20
RAN, 83
reachability graph, 11
receive, 42, 43
records, 27
reentrant, 59, 60
refinement, 106
regular expressions, 11
Reisig, Wolfgang, 14, 15
release, 66
remote procedure call, 29–33, 35, 42, 59, 61, 62, 64, 91, 101, 102
asynchronous reply, 32
remote procedure call with asynchronous reply, 32, 63, 101, 102
remote procedure calls, 101
remote procedure calls with asynchronous reply, 101
replication, 71
resource, 66
move, 103
resource allocation net, 83, 85, 94, 102, 103
resource availability, 102
resource pool, 66, 84, 100
imported, 69
local, 69
resources not available, 80
return transition, 101
RMI, 21
role base modeling, 5
Rossi, Francesca, 71
RPC, 29, 32
run, 12–14, 56

S
scalability, 101
scenario, 92
seamless, 35
seamless visual embedding, 107
selection, 67, 71
liter, 67
port state, 67
port state set, 67
selection expressions, 106
semantic, 49
coordination net, 49
send, 42
separation of concern, 88
sequential replication, 71, 76
server, 28
service calls, 92
service net, 83, 85, 86, 94, 103
set, 7
Shapiro, Robert M., 19
shared, 33, 38, 40, 41, 44, 103
shared connection, 40, 41
shared port
  copy, 103
shared state set step relation, 40
Sibertin Blanc, C., 20
signature, 17, 27, 92
simply enabling, 47
skeleton, 87
SN, 83
software architecture, 5
software engineering, 59
Souissi, Younnes, 21
SQL, 71
standard
  high level Petri net, 15
starvation, 28
state charts, 87
state machine net, 7, 11
state step relation, 40
stateless, 30
subtype notion, 27
subtype polymorphism, 27, 42
subtyping, 28
surjective function, 7
Suzuki, I., 59
synchronization, 104
synchronization condition, 51
synchronize, 106
synchronous, 21, 24, 35, 49, 106
synchronous send, 26, 43, 51
syntactical interface, 28

T
temporary, 29
temporary state, 41
term, 17
termination, 60
textual scoping, 48
throwaway, 32, 101
Tokoro, Mario, 22
total function, 7
trace, 29
trace set, 10, 11
traces, 10, 11
transient, 29, 35
transition, 8, 9
  local, 103
  valid, 50
transition fusion, 19, 20
transition invocation, 19
transition mode, 16, 18
transition pair, 51
  valid, 51
transition refinement, 59
transition substitution, 19
triggered, 103
true concurrency, 22
true concurrent, 12, 39, 66, 71
typed port, 27
typing
  literals, 27
  port, 27

U
usage, 46
usage port, 25, 37
use case, 92
user, 28

V
Valette, R., 59
valid, 18, 50–52
  transition, 50
  transition pair, 51, 52
Valk, Rüdiger, 20
Van Hee, K. M., 20
Vasconcelos, Vasco Thudichum, 22
Verkoulen, P. A. C., 20
view, 15
Vogler, Walter, 59

W
Wegner, Peter, 5, 27
Wirfs-Brock, Rebecca, 5
word, 8, 10
References


Preprints
"Angewandte Mathematik und Informatik"

20/98 - S  N. Schmitz: The fair price in the Cox-Ross-Rubinstein model is conservative.
21/98 - I  J. Lechtenbörger, G. Vossen: Unabhängigkeit von Datenwarenhäusern mit Sternschema
23/98 - S  D. Plachky: Note on Groups of Admissible Location Parameters.
26/98 - N  V.P. Palamodov: Holomorphic synthesis of monogenic functions of several quaternionic variables.
27/98 - N  V.P. Palamodov: Dynamics of wave propagation and curvature of discriminants.
28/98 - S  V. Hoefs: Markov Renewal Theory for Stationary m-Block Factors: First Passage Times and Excess over the Boundary.
29/98 - S  D. Plachky: Relations between the Negative Binomial and Negative Hypergeometric Distribution with Applications to Testing and Estimation Based on Inverse Sampling.
32/98 - S  D. Plachky: Bernoulli Experiments
33/98 - I  M. Weske: Design and Implementation of an Object-Oriented Workflow Management System based on CORBA.