An Introduction to Modular Process Nets

Dietmar Wikarski[1]

TR-96-019

April 1996

1. On leave from Fraunhofer Institute for Software Engineering and Systems Engineering (ISST) Berlin, Germany

An Introduction to Modular Process Nets

Dietmar Wikarski

[Dietmar.Wikarski@isst.fhg.de]

International Computer Science Institute, Berkeley, CA

on leave from

Fraunhofer Institute for Software Engineering and Systems Engineering (ISST) Berlin, Germany

TR-96-019

April, 1996

Abstract

Modular process nets are a graphical and formal notation for the representation of technical and business process models containing concurrent activities. Originally this class of Petri nets was developed for the modeling, analysis, simulation and control of workflows and computer-based process control systems, but it is also suitable for use in all other areas where a formal but comprehensible description of complex processes is needed.

After a description of the basic aims and design decisions for modular process nets, the report gives a brief introduction to low-level Petri nets including different types of transition rules and aspects of the descriptive and prescriptive use of process models.

The main and most innovative points which are explained in more detail are the introduction of a hierarchical module concept for nets and the definition of elementary process nets. The module concept is part of a more general ("object-based") approach to Petri nets allowing several types of abstraction, whereas the main points of elementary process nets are synchronous and asynchronous communication between separately interpreted net instances via events and token passing.

Modular process nets are low-level Petri nets equipped with these module and communication concepts and optionally enhanced by the use of a task concept, a method known from the areas of computer-supported cooperative work (CSCW) and workflow management.

Because the report is aimed at a systematic and easy-to-understand introduction to modular process nets, it provides a precise explanation of this net class which is kept as informal as possible and enhanced by some typical application examples.

Contents

1 Introduction

As one can observe by consideration of many new and expanding areas of information systems research and application (e.g. Business Process Re-engineering, Process Innovation, Workflow Management, Enterprise Integration, Concurrent Engineering, Computer Supported Collaborative Work, etc.), model-based "process thinking" is becoming a more and more usual approach to understanding and changing the processes going on in the real world.

On the other hand, "process thinking" in a more abstract sense has been usual throughout the history of computer science in its various domains. Unfortunately, the means of description used vary strongly from domain to domain and are usually not communicable - in a reasonable time - to persons outside of a certain community. But with the penetration of computer supported communication and cooperation into more and more areas of human life, a growing circle of computer users will need to understand such formal notations (e.g. complex workflows in workflow management systems, negotiation processes for the quality of service in communication networks etc.).

Bearing in mind that the processes to be described are inherently distributed in space and time, the use of Petri net models is probably the best choice for a simple but precise means of communication about these processes. In addition to the static graphical representation of the possible control flow, the so-called "token game" allows the visualization of its dynamics.

In recent years, a great variety of classes of net models has been developed and used. In order to enhance the expressivity of net models, there has been a noticeable tendency towards using more and more complex classes of nets with several kinds of tokens, arc and node inscriptions and other extensions instead of the originally "simple" classes of Petri nets. Such a development is seemingly unavoidable for coping with the complexity of models. But with the increasing complexity of the means of description, one of the most important advantages of the net approach, namely easy communicability for humans, is lost. So the price of the high-level net representation of complex models is usually its (near) non-communicability to those with no experience of Petri nets.

Fortunately, there is one way of obtaining comprehensible compact representations of complex models: modularization and abstraction from the internal structure of modules. These methods have much in common with "zooming in and out" of images, which is familiar to most people from a number of ubiquitous technical facilities (e.g. graphical user interfaces, photo cameras, video-recorders).

Two further problems for most of the existing classes of net models (including the high-level nets mentioned above) are the impossibility of describing changing behavior and the communication of "active"[1] nets with one another and with their environment.

Modular process nets have been designed to meet these challenges. Based on a general object-based approach for Petri nets, they provide a hierarchical module concept as well as constructs for communication between net instances and their environment and for the creation and

1. *i.e. executed by an interpreter or "enacted", as often said in the workflow community. In the following, the term "interpreted nets" will be used.*

destruction of net instances. Although originally developed for human-machine interaction, their use is not restricted to this area. In fact, they allow modeling of any technical processes, especially where aspects of concurrency and distribution have to be included in the model.

Another essential application of the proposed module concept is the natural, local introduction of “time” into Petri nets which leads to a more natural representation of distributed systems using nets and serves as a basis for an integrated qualitative and quantitative analysis of these systems [WiHe95]. Here, the key ideas are the composition of modules from so-called conflict clusters and a quantitative abstraction of the behavior of the modules by so-called DDP (Defective Discrete Phase) distributions [Wika90], [Ciar95]. Though these points are not discussed in this paper in detail, it is important to mention these applications in order to emphasize the generality of the proposed module concept.

The aim of this report is a systematic and easy-to-understand introduction to modular process nets. Emphasis has therefore been placed on providing a precise explanation which is as informal as possible, hopefully not resulting in unnecessary redundancy.

After a brief description of the basic aims and of the design decisions derived from them (Section 2) the usual definitions and some specifics of low-level Petri nets are summarized (Section 3). The module concept for nets is introduced in Section 4. In Section 5, the definition of elementary process nets is given. These two concepts make possible the definition of modular process nets in Section 6 which is closely connected with the concept of tasks as it is usually applied in the CSCW and workflow communities. Section 7 presents some simple but typical examples of modular process nets taken from the areas of workflow modeling and software architecture procedure description in order to illustrate the modular process net approach.

2 Aims and design decisions

In addition to the requirement of compatibility with existing Petri net classes, *process nets* were designed with the following basic objectives in mind. This class of net models should

(1) be simple, easy-to-learn and comprehensible,

(2) allow compact representation of complex processes,

(3) allow automated computer interpretation[1], for enactment of the specified processes,

(4) allow the distribution of its interpretation over different processors, to enable spatially distributed (e.g. work) processes,

(5) be flexible, to allow for changes in the net structure during interpretation,

(6) enable integration with organization models,

(7) be safe with respect to their execution on one or several processors.

These aims gave rise to the following design decisions, which are numbered in the same order as the basic objectives:

(1) Elementary low-level Petri nets are used as the underlying net class, i.e. the tokens are identical and there can be a maximum of one token on each place.

(2) In order to be able to represent *partial nets* of given nets independently of the surrounding net parts, the concept of *net modules* and *module transitions* is introduced. Two concepts are available for their use: one flat concept (*decomposition into* and *composition of* net modules) and one hierarchical (*coarsening to* and *refinement of* module transitions). Module transitions on a given hierarchical level can be used to represent net modules of a lower level.

(3) Generally, the *immediate firing rule* is assumed to be applied, stating that a transition *must* fire as soon as it is enabled [2]. The *may firing rule* (stating that an enabled transition *may* fire, but need not do so) can be simulated by using so-called *sensor transitions* which are a special kind of *event transitions* (cf. point (4)).

(4) For the communication between different nets, which may even be executed on different processors, two types of mechanisms are introduced: *message-based communication* (based on *interface places* shared by different nets) and *event-based communication* (based on *event transitions* in different nets which know the same type of *events*).
Interface places enable asynchronous communication between nets. Tokens which are placed on an interface place of a net are also available at all places bearing the same name, even in nets which are interpreted separately.
Events can be triggered by *trigger transitions* as well as by other software processes.

1. i.e. execution on a interpreter

2. A transition is enabled when all its preplaces are marked.

Sensor transitions fire when and only when they are enabled and an event assigned to them occurs[1]. In addition to these types of event transition there are *set-alarm transitions* and *clear-alarm transitions* which signal events at a certain time and delete them beforehand, respectively and *PN-start transition*s and *PN-end transitions*, which are responsible for generating and removing net instances.

(5) In view of the aim of achieving flexibility, in particular for the implementation and execution of *workflow models*, the concept of *tasks* including the *call concept* has been introduced in the form of *activities*, which may be considered as elementary "task" module transitions. *Activities* are a special kind of module transitions whose refinements communicate with other parts of a process model via events. In a workflow management system based on process nets, tasks form the interface for the user's interaction with the interpreted nets.

(6) The essential link for the connection of organization models and process models is provided by *roles* which define the responsibilities and authorizations of *persons* or *groups of persons* with respect to the completion of tasks. Roles or persons can be assigned to the tasks and activities as parameters.

(7) Process nets are designed to be *safe*, i.e. with a maximum of one token on each of their places. Refinement, fusion and calling of safe process nets should result in further safe process nets.[2] Furthermore, a restricted *liveness* property should apply: for every process net started, a PN-end transition (cf. Section 5.1) must become enabled at some point.

The class of *modular process nets* (cf. Section 6) is constructed out of the basic class of *elementary process nets* (cf. Section 5) using the *module concept* of process nets (Section 4). The main reason for this division into two separate classes is the intention to interpret the nets using interpreters for elementary process nets, but to use modular process nets for communication with the user. For the latter, tasks are captured by the module transitions called activities (cf. point (5)) which are refined on the level of elementary process nets to special net modules.

The *syntax* of elementary process nets corresponds to that of ordinary Petri nets which have been extended by introducing additional types of transitions (cf. Section 5.1). For modular process nets which are to be used for workflow management purposes in the form of process modules (cf. Section 6.2), some syntactical restrictions are required. On the level of modular net representation, this leads to conflict-free nets.

As in any class of Petri nets, the *semantics* of process nets is locally defined by *transition rules* (Section 3.1). The *pragmatics*, i.e. the purpose and means of using a process net model, depends on the intention of the user. Process nets can be used as both prescriptive and descriptive models. This distinction is discussed in Section 3.2.

1. *The may firing rule, which is standard in net theory (a enabled transition can fire but does not have to), can thus also be simulated by sensor transitions.*

2. *This property requires further research which may make it necessary to restrict the net class with respect to the possibilities for these operations.*

3 Petri nets

Nets are bipartite directed graphs, i.e. directed graphs where each node is assigned to one of two types and where adjacent nodes are always of different types. For visualization, one of the types is generally represented by round symbols and the other by rectangular ones (including bars). In the following the round symbols will be referred to as *places* and the rectangular ones as *transitions*, these being the most usual notations.

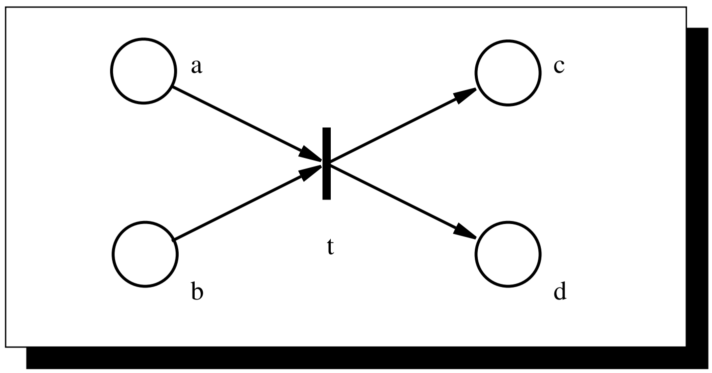

Fig. 1: Example net with places a, b, c, d and transition t

Nets serve as a basis for various classes of models (“net classes”) with which the behavior of distributed systems can be represented graphically and with formal rigor.

A net can be used, for example, to model the control flow in a distributed system. Here, two types of branches and confluences of the control flow can be explicitly modeled: start and completion of parallel threads of control (corresponding to the logical “and” and associated to branching out of and into transitions) on the one hand, and start and completion of alternative control paths (corresponding to the logical exclusive “or” and associated to branching out of and into places) on the other.

A *Petri net* is a net with the following information assigned to its nodes:

- The *places* can be marked with *tokens* (represented graphically as black dots) to denote variable local states.
- The *transitions* are assigned a *transition rule* which locally defines the changes in the markings of the places adjacent to them.

The terms (*global) state* or *marking of a net* are used to refer to the numbers of tokens on the places of the net at a given point in time. The term *local marking* is used for a given transition to refer to the current marking of the places attached to this transition by incoming or outgoing arrows.

By assigning a transition rule[1], the system modeled by the net is assigned a non-deterministic behavior whose characteristics (e.g. the set of reachable global states, freedom from deadlocks, invariants[2]) can be derived from analysis of the net structure together with its initial marking and the transition rules.

A visualized simulation allows the behavior of the described processes to be understood and predicted. Computer *interpretation* of Petri nets (also called *execution* or *enactment*) can be used to control real processes associated with the interpreted nets.

3.1 Transition rules

The assignment of a *transition rule* to the transitions results in the assignment of *behavior*[3] to a net. Generally, it states that any enabled transition can fire.

Firing of a transition means that tokens are removed from each of its preplaces (i.e. those places from which arrows point to the transition) and tokens are put on each of its postplaces (i.e. those places into which arrows point from the transition). Under the conditions being considered here (i.e. neither predicates nor multiplicities are assigned to the arrows) exactly one token is removed from each preplace and exactly one token is put on each postplace.

The *enabling rule* of a transition determines the conditions under which a transition *is enabled* to fire (an equivalent term is: when it has *concession*).

3.1.1 Enabling of transitions

Regarding the first aspect of a transition rule (enabling), a distinction is made between the *normal* and the *safe* enabling rule. In the case of the *normal enabling rule* a transition is enabled for a given local marking when and only when all of its preplaces are marked. In the case of the *safe enabling rule* it is also required that all of a transition's postplaces are not marked.

A net is said to be *safe* if the structure of the net guarantees that when using the normal enabling rule there will never be more than one token on any place. Safe nets thus show the same behavior under both enabling rules.

For process nets it is required that every instance (i.e. every interpreted process net) is safe. Thus, the interpreters can use the normal enabling rule but the nets will behave as if the safe enabling rule has been applied.

---

1. *which is usually set identical for all transitions of a Petri net*
2. *see, e.g.[Reis85] or [Star90]*
3. *Note that the behavior of a net is defined in fact locally (by the behavior of transitions) but usually naturally extends to a global one (by the connectedness of the net).*

3.1.2 Firing of enabled transitions

If a transition is enabled, it *may* fire at *some time* or *must* fire *immediately.* The distinction between these two cases is reflected in the distinction between two kinds of firing rules, the *may firing rule* and the *immediate firing rule.* Note that in the case of the may firing rule, an enabled transition may lose its enabling to other enabled transitions which "steal" tokens from the preplace in question when they fire. Thus, the global behavior of a net with the may firing rule is generally "richer" than the behavior of a net with the same structure, but which uses the immediate firing rule. Let us consider these two rules in more detail.

May firing rule:

Any enabled transition may fire but it does not have to do so.

Assuming this to be the case, one can find out all the global states which the net can ever reach. The graph which is characterized by all the reachable global states of a Petri net (as its nodes) and the corresponding state transitions (as arrows) is called the *reachability graph* of this net and the analysis of the net's behavioral properties based on this graph is called *reachability analysis*.

Immediate firing rule:

Any transition must fire as soon as it becomes enabled.

The application of this firing rule implies that maximal sets of enabled transitions have to be fired. Therefore, it is necessary to know the global marking at any point in time in order to find these sets. Another consequence of the firing of maximal sets of transitions is that certain markings reachable under the may firing rule are no longer reachable. For this reason it is not possible to simply transfer the results of the reachability analysis for a net with the may firing rule to a net with the same structure whose transitions fire according to the immediate firing rule.

3.2 Descriptive and prescriptive models

The relation between a *net model* (a particular kind of behavior model) and the modeled real system can vary. On the one hand, the behavior model can be a description of the observed or assumed behavior of a real system. We refer to such models as *descriptive* ones. On the other hand, a behavior model can be considered as a specification or program for the future behavior of a real system. In this case we refer to the model as *prescriptive*.

In the case of net models, there is a close connection between these two kinds of behavior model and the two types of firing rule described in Section 3.1.:

The *may firing rule* corresponds to the descriptive pragmatics of net models, because it reflects the local way of proceeding both with regard to obtaining a net model of reality from observations and for reasoning about the *possible* behavior of a distributed system described by a net. The decisions concerning the “resolution” of conflicts between transitions, i.e. the selection of one transition to be fired out of a set of transitions with concession, do not have to be explicitly stated. Instead, all the possibilities are non-deterministically included in the model. In addition to being well suited for interpersonal communication (communication about and understanding of all possible processes), these conditions are also appropriate for computer-based behavior analysis (finding deadlocks and unsafe situations). However, simulations should be carried out manually (by playing the token game) or with computer support. In the general case (i.e. where conflicts between transitions are possible) they can not be fully automated - in view of the conflicts to be solved by ‘hand’.

The may firing rule is *not* suitable for the automated *interpretation* of Petri nets for process control, because it neither guarantees progress in an operational model, nor does it prescribe how to proceed in conflict situations. In such cases, the *immediate firing rule* should be used and the resolution of conflicts should be ensured by suitable mechanisms. Thus, the progress of an interpreted net is ensured (in cases where the net is live in each reachable marking) as well as the unique choice of a set of transitions (enabled and not in conflict with one another) to be fired in every reached marking.

4 Module concept

4.1 Introduction and summary

One of the aims of the introduction of modules into Petri nets is to be able to represent large or complex nets in a more manageable form. This can be done by separation into and by abstraction from the detailed structure of *partial nets*[1]. Obviously, partial nets should be able to be further separated, shown individually and joined together. Moreover, we wish to represent partial nets as *modules* (in the sense of software engineering) in order to gain fundamental advantages in process technology similar to those brought to the software life cycle by the use of software modules. These advantages include re-usability of modules, simplified testing and separation of areas of responsibility.

The main feature of the concept of the *net module* which most distinguishes it from that of an "ordinary" partial net is the explicit definition of *interfaces* which define the possibilities of interaction of a net module with its environment. *Abstraction* from the module's internal structure and *refinement* of module "frames" by enriching them with structural and/or behavioral information are two further important aspects of a well applicable module concept for nets.

Given the assumptions as described above - i.e. modules are defined or generated by partial nets - there are at least three possibilities for defining interfaces so that the composition of net modules can be done in a well-defined way:

(1) The interfaces are defined by *fusion nodes* which are part of both modules. Composition of modules is done by fusion of corresponding fusion nodes, i.e. of those with identical names. Decomposing a net into partial nets leads to redundant fusion nodes, i.e. to fusion nodes with the same name in different partial nets (see Section 4.2)

(2) The interfaces are defined by *arcs* connecting nodes of different modules. Because nets are bipartite graphs, these interfaces will be heterogeneous in the sense that the type of every output node will be different to that of the corresponding input node.[2]

1. *A partial net is defined as such a part of a net which is characterized by a subset of nodes of the original net and which contains all the arrows between these nodes in the original net.*

2. *For nets where the normal enabling rule is applied (see Section 3.1.1), a consistent heterogeneous interface concept has been developed in the context of object nets. In this concept, the output nodes are transitions and the input ones are places ([Wika90], [HeWi95])*

(3) The interfaces are composed of
- sets of (input and output) *port transitions* which are part of the modules and which are referred to in the following as *input ports* and *output ports*, and
- sets of *interface places* ("buffers") each of which is connected with an output port of one module and with an input port of another one.
In contrast to option (1), both interface places and the arcs connecting them with the input and output ports of modules are pure interface elements not belonging to any of the modules like the interface arcs in option (2). Note that in contrast to option (2), the interfaces here are "homogeneous".

In the module concept proposed in the following, a combination of the options (1) and (3) will be used, i.e. we assume the modules to be transition-bordered subnets where the transitions of a module connected with places outside a module are considered as its *ports*.

The use of net modules as a special kind of transition-like nodes allows a natural notion of refinement and abstraction in nets: Given a net module symbol, it can be refined to a partial net which expands the original net structure. On the other hand, the module symbol can be used as an abstract representation of a given partial net. If the module symbol also contains symbols for port transitions of the abstracted net, the arcs connecting these port symbols with the surrounding interface places have the same semantics as in the refined expansion of the net, i.e. every set of arcs connected with one port symbol implies an "and" relation for the places connected with these arcs (see Fig. 9 in Section 4.3.2.).

The interface places used for the composition of modules can be interpreted as ordinary places of a new class of nets. In this net class ("Modular Process Nets", see Section 6), in addition to "ordinary" places and transitions, the new node type "module transition" can be used (as the representation of transition-bordered partial nets, see Section 4.3). For a unique unfolding of nets containing module transitions with separately given refinements it is necessary to represent these refinements together with the interface places. To make clear that these interface places are redundant, they are highlighted on the refinement level as *fusion places* (see also Section 5).

For ease of use and for modules where interfaces have not yet been defined - e.g. for top-down design of systems - we also allow a semantically "weaker" representation of net modules, called *representation without interfaces* (see for example Fig. 8 in Section 4.3.2). Here it is only known that the represented partial net is transition-bounded. The arcs connecting such module transition symbols have a restricted semantics compared with those of nets without module symbols.

Semantically stronger abstractions have been developed in the concepts of *Markovian object nets* (see for example [Wika90], [WiHe95]) and of *"logical" object nets* ([Whit93]). These concepts allow semantic abstractions of partial nets and net modules by formalisms beyond the scope of Petri nets (Markovian processes and DDP distributions [Ciar95] on one hand and temporal logics on the other) and will not be considered in more detail in this report. It is only important to note that both concepts are based on a concept of modules similar to that described here.

In the following sections, the module concept for process nets will be introduced in more detail.

4.2 Decomposition and composition of nets

Let us assume that for a net a *decomposition* is given, i.e. the set of its nodes is decomposed into disjoint subsets. This permits a complex net to be represented by showing separate net modules which can, in this case, be considered as “sections” or “views” of the net, according to the abstraction principle of *information neglection* known from software engineering. In order to be able to reconnect the net modules defined by a decomposition, unique interfaces are necessary. These interfaces are represented by *ports* and *fusion places* while partial nets become net modules. For a given partial net of a net, the fusion places are those places in the partial net which are connected with transitions outside the partial net and those places outside the partial net which are connected with transitions inside the partial net[1]. The ports of the net module corresponding to the partial net are those and only those transitions which are connected with fusion places, where the output ports are the predecessors of fusion places and the input ports are their successors. In order to be able to achieve a unique composition of the net modules of a given decomposition, all the fusion places connected with the net modules are included in the representations of each of them, i.e. redundantly (cf. the example in Fig. 2).

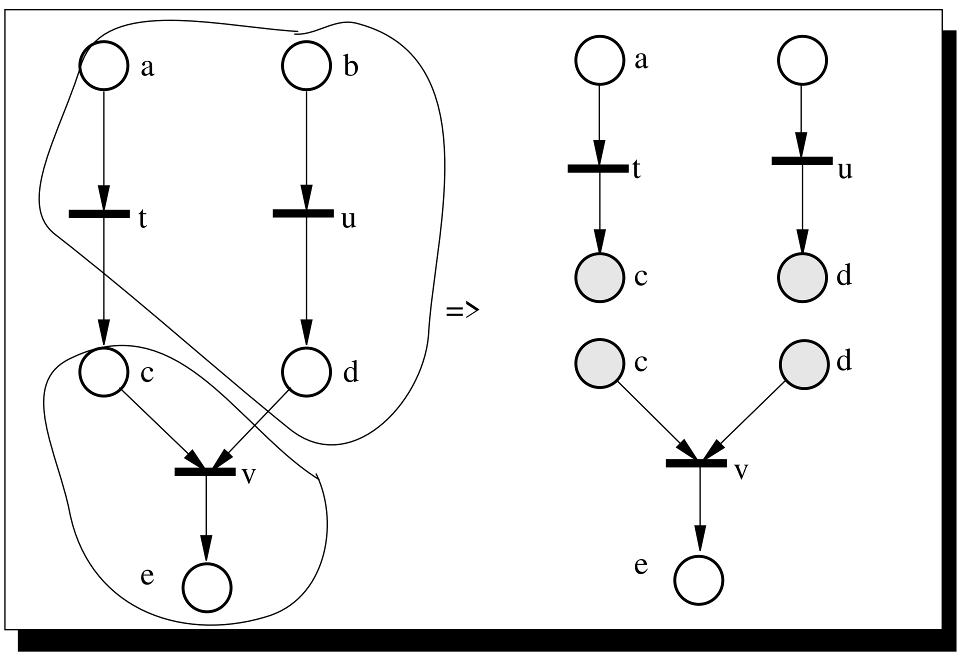

Fig. 2: Decomposition of a net into net modules

1. *This definition is compatible with that of Baumgarten [Baum90] in which the (relative) boundary of a partial net with regard to an overall net is taken to be the set of those nodes which are connected by arrows with the remaining net. The absolute boundary of a net consists of those nodes of a net which have either no predecessor or no successor. The term is also compatible with the fusion places used in Section 4.3.2 for refining module transitions by transition-bounded nets.*

The composition of several separate partial nets which have fusion places with the same names into one new composite net is carried out by replacing all the fusion places bearing the same name in all the various partial nets with one new place of that name which is no longer shaded.

In Fig. 3, which shows the (re)composition of the decomposition shown in Fig. 2, the transitions t and u are the output ports of the one module and v is the input port of the other, while c and d represent the common fusion places.

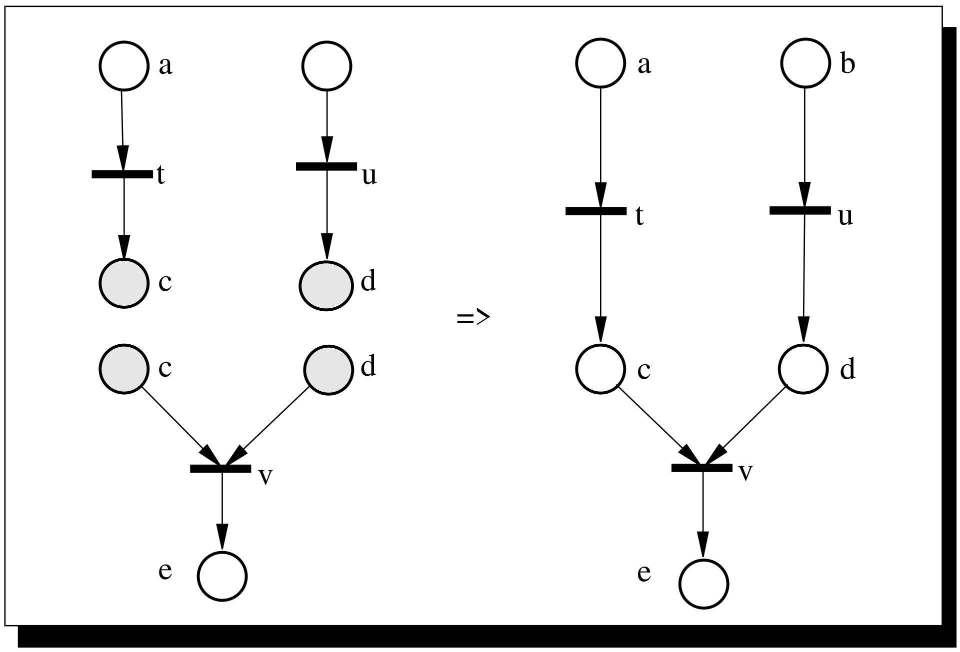

Fig. 3: Composition of net modules

The operations *restriction* and *embedding* introduced by [Baum90] can be considered as special cases of the decomposition (information abstraction) and composition (information enrichment) operations with regard to a specific partial net. Restriction means to decompose a net into an "inner" and an "outer" (surrounding) net and embedding means to compose a new net out of a given net and its surrounding. The difference between the two pairs of concepts is that in the case of restriction and embedding each of the nets is given a special "role" (to be an "inner" or "outer" net), whereas with decomposition and composition all the nets involved are considered to be equal.

4.3 Coarsening nets and refining module transitions

In contrast to the decomposition and composition of nets and (partial) nets as described in the previous section, we now assume that net modules can be encapsulated and that abstractions can be made from the contents of a module, which corresponds to the abstraction principle of *information hiding* in software engineering. An abstract module representation, which is referred to as a *module transition*, is used to abstract from the internal structure of a module but also to give interface information which characterizes the behavior of the module with respect to its environment. This is achieved by defining sets of input and output ports. For module transitions created by the decomposition of a net into net modules, these ports must be connected to the fusion places in the same way as the transitions of the modules from which they were derived by decomposition.

The two procedures - coarsening and refining - which can be realized using this module concept are described below. Applied to system design, these two methods correspond to the “bottom up” and the “top down” approaches, respectively.

4.3.1 Coarsening transition-bounded partial nets

Assume a transition-bounded partial net is to be coarsened into a module transition. Those places in the surrounding net which are connected to transitions of the partial net are marked as fusion places. These transitions are referred to as port transitions or, for short, *ports* of the subunit. Where arrows run from a fusion place to a port, that port is called *input port*, for arrows from a port to a fusion place the port is called *output port*. Ports which are both input and output ports are not permitted.

The net is now decomposed into two levels: In the “upper” coarse grain level the chosen partial net is replaced by a *module transition symbol.* All arrows from the fusion places to the input ports and from the output ports to the fusion places are connected to the module transition (cf. Section 4.3.1.1 and Section 4.3.1.2.). The “lower” refinement level consists of the chosen partial net with the marked fusion places.

In order to make possible both the visualization of module interfaces in accordance with the module concept and the comprehensible representation of nets with module transitions, two representation modes for module transitions are offered: *with interfaces* and *without interfaces*.

4.3.1.1 Module transition representation with interfaces

Here, the ports are part of the module symbol. Their number therefore depends on the structure of the partial net to be represented. Input ports are symbolized by unshaded boxes, output ports by black boxes. Whether a pair of incoming or outgoing arrows are in an “and” or an “or” relation to one another can be decided given the net on the coarse grain level. If two arrows belong to the same port, then tokens are fired through both of them together into (in the case of an input port[1]) or out of (in the case of an output port) the module transition. This corresponds to an "and" relation between the arrows. If the arrows belong to different ports, then no statement can be made about the possibility of simultaneous firing. This corresponds to the "weak" (not exclusive) "or" relation.

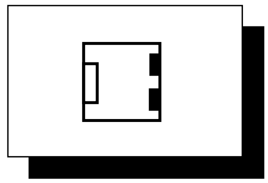

Fig. 4: Representation with interfaces of a module transition with one input and two output ports

If all outgoing (incoming) arrows are in an exclusive-or (="exor") relation, i.e. if all output ports have a common preplace or all input ports have a common postplace, this can be denoted by including a switch symbol in the module symbol. This constitutes a further refinement in the representation of module transition interfaces compared to that of ports.

Thus, for example, the symbol shown in Fig. 5 means that the two output ports have a common preplace and the symbol shown in Fig. 6 means that the two input ports have a common postplace.

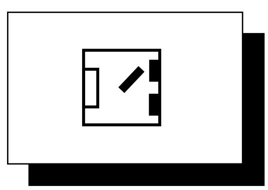

Fig. 5: Representation with interfaces of a module transition with two alternative output ports

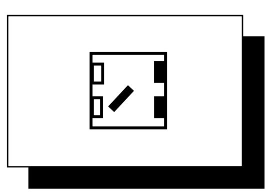

Fig. 6: Representation with interfaces of a module transition with two alternative input ports

1. In cases where several arrows go into one port, this may involve waiting for synchronization.

4.3.1.2 Module transition representation without interfaces

In this representation mode the ports are omitted from the transition symbol. This may be for reasons of easier comprehension, for instance if there are too many ports on a module transition, or because the number and/or type of connections of the ports with the surrounding nodes has not yet been determined. Graphically, the representations without interfaces of module transitions consists of double-bounded transition symbols.[1]

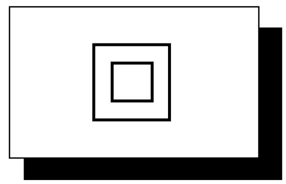

Fig. 7: Module transition representation without interfaces

One advantage of representation with interfaces of module transitions over representation without interfaces is that since the ports are included, all the arrows which are connected to the fusion places are also shown. This means that all the fusion places are always “complete” on the coarse grain level (including all incoming and outgoing arcs) so that they require no special attention on this level. As an example, consider the arrow from place “a” to transition “y” in Fig. 8 and Fig. 9 in Section 4.3.2: Whereas this arrow is not visible in the module transition representation without interfaces (Fig. 8), it is well represented in the module transition representation with interfaces (Fig. 9).

4.3.2 Refining module transitions

The module transitions presented in Section 4.3.1 make it possible to define nets which are not yet completely specified. Such an approach is typical for top down design, assuming that module transitions may be further refined.

For such a refinement, all places which are connected to a module transition on the coarse grain level are represented on the refinement level below as fusion places. This representation makes clear both the redundancy of representing this place - it appears on the refinement level a second time - and the fact that not all the incoming and outgoing arrows of these places are present on the refinement level[2]. If the module transition is represented with interfaces, then the ports on the refinement level are identified with the transitions connected to the fusion places.

1. *Although this is very similar to the usual Petri net transition symbol, in the case of representation without interfaces the groups of incoming or outgoing arrows are neither in an “and” relation (as is the case with transitions) nor in an “or” relation (as is the case with places). Groups of arrows which are connected to one fusion place on one side and to several ports on the other are therefore also represented by exactly one arrow.*

2. *In the case of representation without interfaces a different type of incompleteness on the top level is possible, see the arrow from place a to transition y in Fig. 8.*

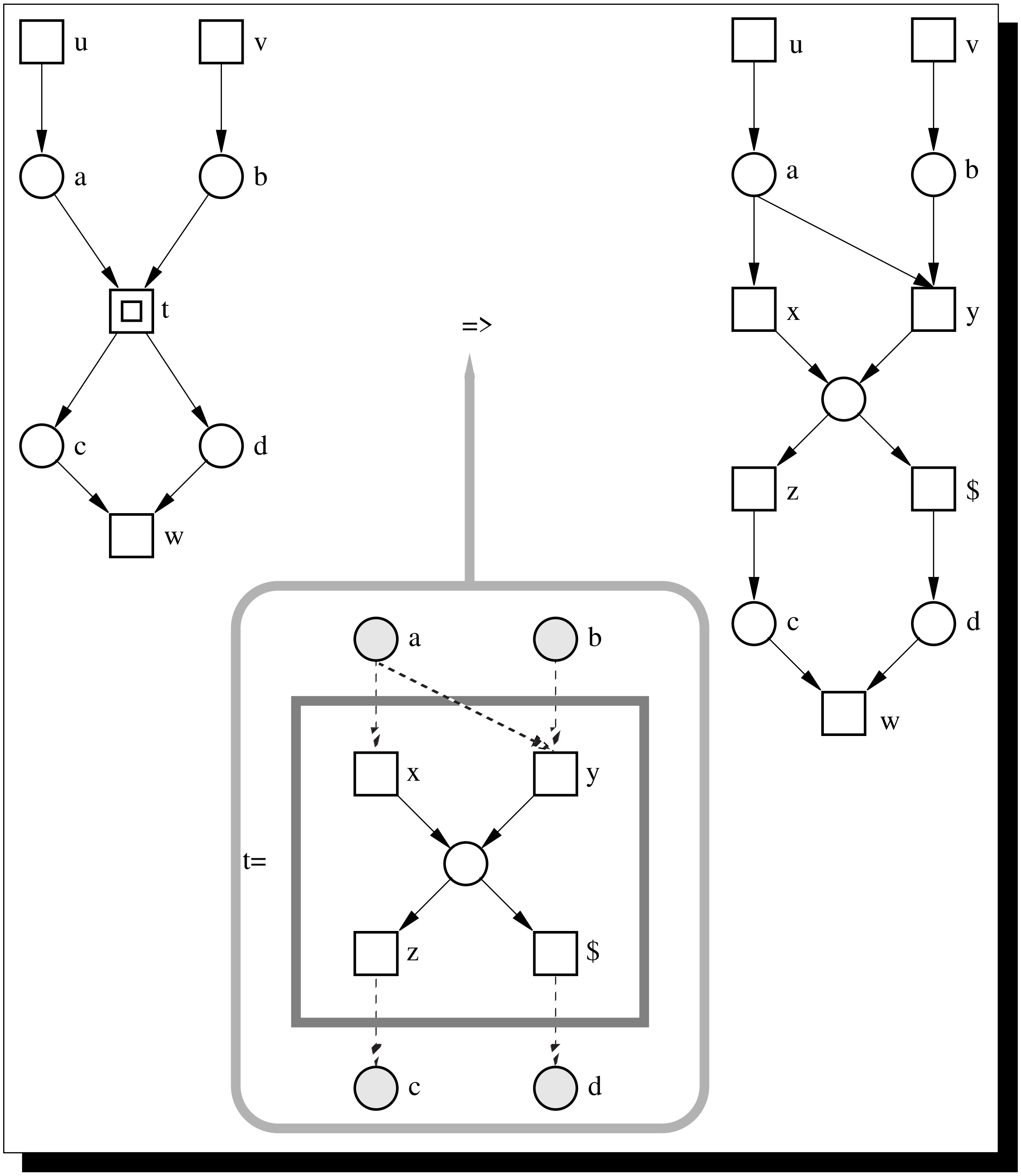

Fig. 8: Representation without interfaces of module transition refinement[1]

1. The representation of the refinement corresponds to the representation principle of the MoPEd tool [HaWi95]: The arrows between fusion places and port transitions are dashed in order to reflect the impossibility of editing these graphical objects on the refinement level.

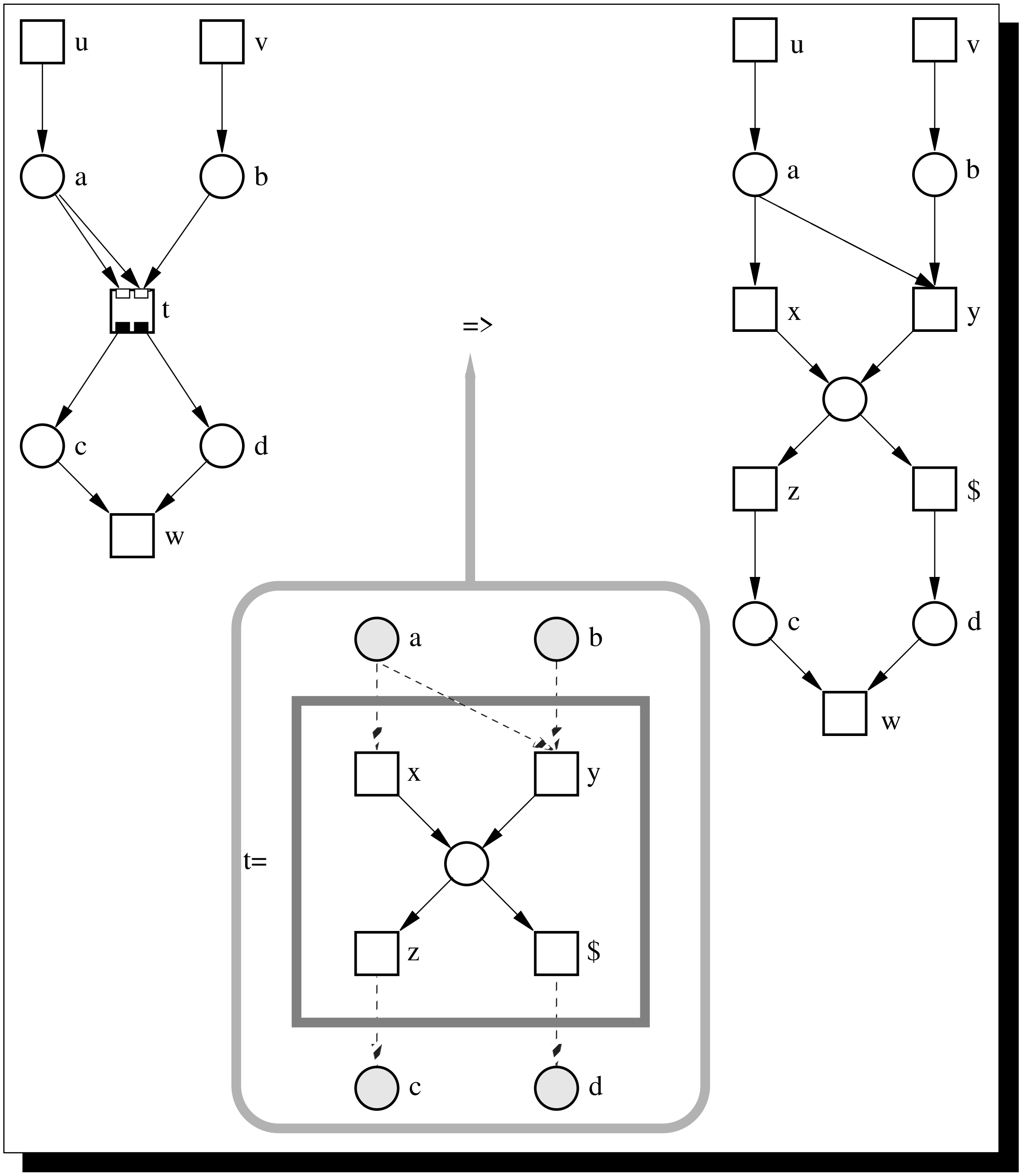

Fig. 9: Representation with interfaces of module transition refinement

4.3.2.1 Context conditions for refinement

By specifying the surrounding net of a module transition, and if necessary its input and output ports, context conditions are defined for the net to be refined, thus restricting the structure of this net.

If there is a representation without interfaces of a module transition, then the numbers of preplaces and postplaces of the refining net can be derived from net representation on the coarse level. However, this does not determine the numbers of arrows coming into and going out of the refining net. All that is known is that the number of these arrows is greater than or equal to the number of preplaces or postplaces, respectively.

If there is a representation with interfaces of a module transition, then the numbers of input and output ports can also be derived, as well as the specification of all arrows connecting the ports to the surrounding places.

If a representation with interfaces contains “exor switches”, then even more information is available, as this implies the existence of a common preplace and/or a common postplace for all output and/or input ports.

4.3.2.2 Marking module transitions

So far, module transitions have been examined from a structural point of view. If the marking of a flat net is to be included even in its hierarchical representation with module transitions, then it must also be possible to visualize the marking of a partial net represented by a module transition. It is therefore useful to agree on a graphical convention (e.g. raising the outline or coloring the surface of the symbol) denoting the case that at least one of the transitions of the represented net has concession.

5 Elementary process nets

As required by the basic objectives (3) and (4) in Section 2, the class of process nets to be designed should allow automated computer interpretation with the possibility of distribution on several, interconnected processes which are interpreted by various processors at different locations.

Because these requirements are independent of the aims leading to the module concept (see Section 2, point (2)) and because the aim is to interpret only *unfolded* (i.e. non-hierarchic) nets, the class of process nets satisfying requirements (3) and (4) but not requirement (2) will be defined in this section as *elementary process nets* separately from the more general class of *modular process nets* which will be introduced in Section 6.

The syntactic extensions of elementary process nets as compared to (elementary) Petri nets consist in the availability of new *node types* (as subtypes of places and transitions) which can be used in particular for communication between interpreted nets and their environment. This "environment" may consist of other interpreted nets, software processes or even users.

More precisely, the structure of an elementary process net is a net (cf. Section 3) in which each place is assigned to one of three types and each transition to one of seven types. These node types are introduced in Section 5.1 below. The semantics of elementary process nets follows from the meaning of the node types explained in Sections 5.2 and 5.3.

5.1 Node types

Each place in an elementary process net is assigned to exactly one of the following node types:

- (ordinary) place
- fusion place
- channel.

Fusion places (see also Section 4) and channels are also referred to as *shared places*. Fusion places are used to improve the graphical representation of large nets. Channels are used to support communication between nets via token passing.

Each of an elementary process net's transitions is of one and only one of the following types:

- (ordinary) transition
- trigger transition
- sensor transition
- set-alarm transition
- clear-alarm transition
- PN-trigger transition

- PN-start transition
- PN-end transition

The last seven of the transition types deal with the handling of events and are also referred to as *event transitions*. *Events* are globally visible and recognizable occurrences which can be both generated and registered by both a process net and its surrounding net. They permit synchronous communication between a running process net and its surrounding net as well as between various process nets running simultaneously.

According to Petri net conventions, places and transitions are represented using circles and boxes or bars. For ordinary transitions, the graphical representation of a transition as a square box can be used to make clear that this transition represents an *action*, i.e. it may seemingly “contain a token” for a non-specified amount of “time” in contrast to transitions which are represented as bars and which stand for “timeless” changes. Fusion places are shown as circles with a bar, channels as shaded circles and event transitions as boxes with special arrow symbols. Unlike the names of ordinary places and transitions, it is obligatory to state the names of fusion places and channels as well as the names of events, since these characterize process net interfaces.

As a new kind of inscription into transitions, angle-like symbols have been introduced for the representation of events connected with transitions in the following way: An angle whose apex touches the “output side” of a transition[1] denotes the triggering of an event. If the legs of the angle touch the “input side” of a transition[2], this transition “waits for” and recognizes events and is called *sensor*. If the transition symbol contains a second angle pointing in the same direction, this indicates the generation of a net instance. The two resulting cases - “apex-touching” and “leg-touching” - correspond to trigger and sensor transitions and are called PN-trigger and PN-start transitions respectively (cf. Section 5.2). If a trigger transition has a second angle pointing in opposite direction - i.e. a rhombus inscribed in a square - this indicates the deletion of the process net to which this transition belongs.

An overview of the node types used in elementary process nets including their graphical representation is given in Fig. 10.

1. *As a recommendation for the graphical representation of process nets, all outgoing arcs of a transition should be connected to only one side - the ”output side” - of a transition, which is the lower one where top-down is the main flow direction or the right one where left-to-right is the main flow direction.*

2. *Analogously, the input side - the upper or right side of the square - should be connected with all incoming arcs.*

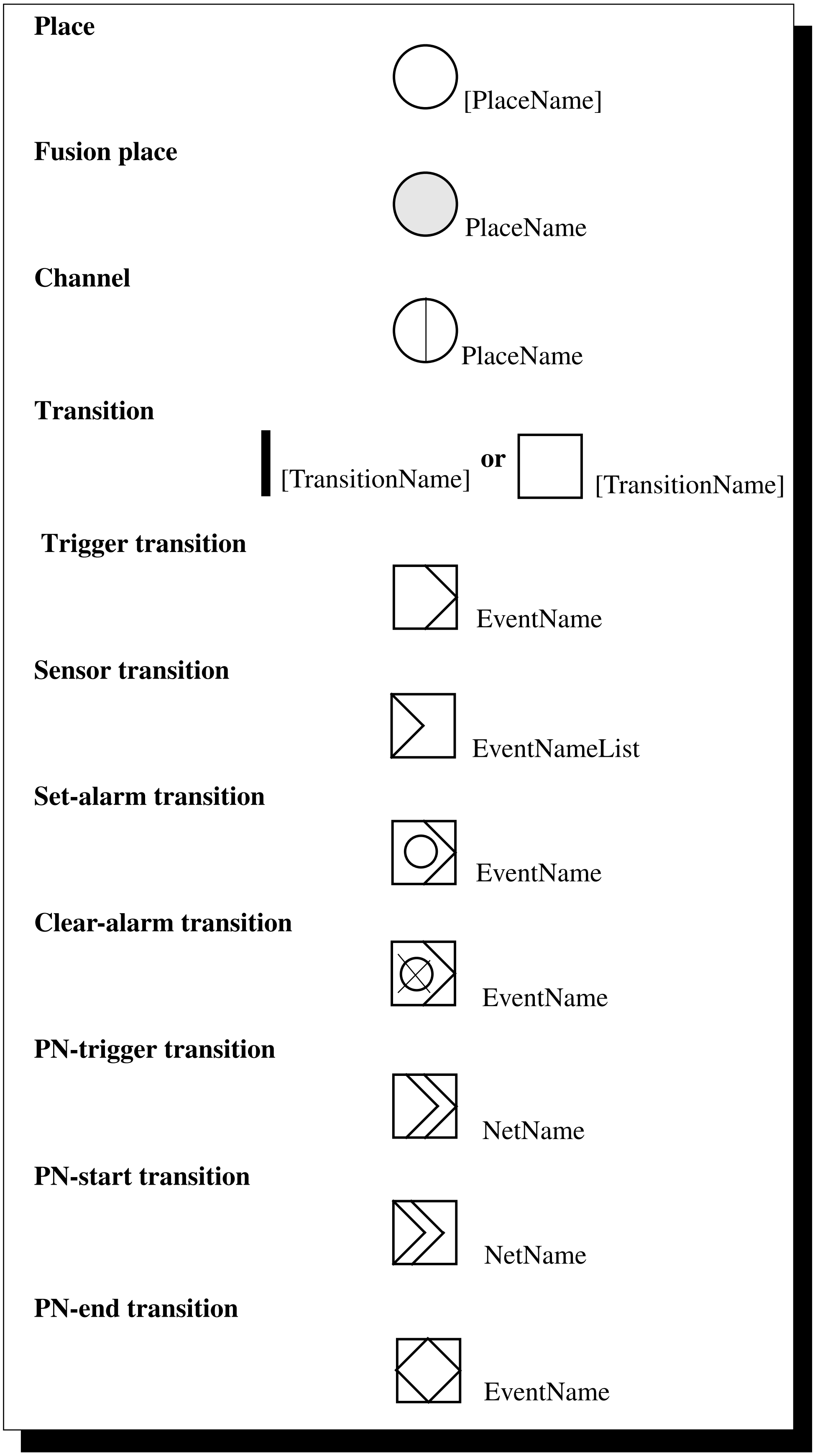

Fig. 10: Node types for elementary process nets

5.2 Transitions: Firing rules and event communication

Due to the necessity of an automated simulation of elementary process nets, the *immediate firing rule* is used for all types of all transitions, i.e. any transition must fire as soon as it is enabled. Note that if transitions represent *actions* - this can be made clear by using the box symbol instead of the bar one (see Section 5.1) - tokens may appear on the postplaces of such transitions only after a (unspecified) delay.
The *event concept* of process nets leads to several modifications of this basic rule for the different types of transition. The main aim of this concept is to enable synchronous communication between a running process net and its environment, where the communication between different running nets is a special case. A detailed description of these modifications is given in the following:

(1) At the moment when a *trigger transition* fires, a globally visible event of the type EventName is generated.
This can influence the progress of interpretation in the surrounding software, including other interpreted process nets with a sensor transition where EventName occurs in EventNameList (cf. (2)).

(2) A *sensor transition* fires when and only when it is enabled and an event associated with it occurs. If it doesn't have concession, there is no effect.[1] By definition an event is associated with a sensor transition if its EventName appears in the sensor transition's EventNameList.

(3) A *set-alarm transition* triggers an event of the type EventName which occurs after a delay specified by the transition.

(4) A *clear-alarm transition* prevents the signaling of a delayed event of the type EventName (cf. (3)). If the clear-alarm transition only fires after the event has occurred or if no suitable event has yet been initiated by a set-alarm transition, then the transition fires like an ordinary transition and events have no effect.

(5) The firing of a *PN-trigger transition* triggers an event of the type NetName leading to the generation of a new net instance of the type NetName (see point (6)).

(6) The occurrence of an event of the type NetName which is associated with a *PN-start transition* generates a new instance of the net to which it belongs and causes the firing of this transition. Since process nets are always generated only by events, every executable process net must contain at least one PN-start transition.

(7) The firing of a *PN-end transition* terminates the net instance in which this transition is situated. At the same time an event of the type EventName is triggered whose name informs its environment which transition's firing caused the net to terminate. (This is important in cases where there are several end transitions.)

In modular process nets, the event concept is also used for resolving conflicts (cf. Section 6).

1. *Since an event can occur at any time, sensor transitions can be used to simulate the may firing rule.*

5.3 Places: Local states, merging and asynchronous communication

As is generally the case in Petri nets, *places* serve to represent local states of the system being modeled. If a place is marked, this signifies the current validity of the condition given by the name of the place[1]. (This applies to all the possible types of places in process nets, i.e. ordinary places, fusion places and channels.)

The common feature of *shared places* (i.e. the fusion places and channels) consists in their potential incompleteness regarding the numbers of incoming and outgoing arrows. The sum of a shared place's incoming and outgoing arrows is equal to the union of the numbers of incoming and outgoing arrows for all the shared places bearing the same name. For increased clarity of a graphical representation, the ("logical") place in question can be considered as a fusion of all shared places bearing the same name.

The difference between fusion places and channels consists in the following: *fusion places* can be used for simple representation of complex nets, these nets must all run on a common interpreter (processor). *Channels*, on the other hand, can be used to represent interfaces between various, separately interpreted process nets. Unlike event transitions, channels support asynchronous communication, i.e. if a token is placed on a channel, it is not necessarily immediately visible to the transitions at the outgoing arrows. On the other hand, this *type of communication* is *safe*, since tokens - unlike events in the case of transitions without concession - cannot get lost. The disadvantage of this "safe communication" consists in the possibility that the resulting linked *nets* will be *unsafe*[2]. If there are several process nets with a shared channel, it is possible that the channel of a receiving net receives tokens from sending nets without the following transition having fired between the arrivals of tokens and thus the net may become unsafe or even unbounded.

1. *Note that this is true for safe nets, i.e. for nets where at most one token can reside on any given place. In the case of "ordinary" Petri nets where several (black) tokens are allowed to be on one place, this can be interpreted as multiple validity of a single condition or as a number of indistinguishable objects. In the case of high-level nets, there may be several colored tokens or different objects on one place.*

2. *Note the striking difference between safe communication and safe nets: The former concerns the non-loss of ("transmitted") tokens whereas the latter concerns the limitation of the number of tokens on a place.*

6 Modular process nets

Based on the concepts explained in the previous sections, the complete class of modular process nets will be introduced below. As a shorthand notation, this class of models can be defined as

modular process nets = elementary process nets + module concept

In view of their main application area - the modeling and management of workflow processes - modular process nets have been enhanced by the concepts of *task modules* and *process modules*. Note that modular process nets can be - and have been - applied without these concepts. But in view of their canonicity, of the existing tools and of the ease of their use they will be presented here in more detail. The use of these concepts for the modeling and enaction of workflow processes and for the specification of the interaction of software modules will be illustrated by the examples given in Section 7.

6.1 Task modules

In the context of advanced workflow management concepts, *tasks* form the frames in which the various kinds of cooperative human activity are embedded. A task is set to an individual or to a group for processing. Here, the *role* of the person to whom a task is set must be compatible with the role which is an attribute of the considered task.[1]

In the WAM workflow management system [Adam95] which modular process nets were originally developed for, tasks are set either directly by an authorized person (including the performer him/herself) or by a running process net. In the latter case, the task is associated to a *task module* (see below) of the process net.

The *processing* of a task can either be either done “by hand” (including delegation to another person) or supported by starting one or more process nets. Once started, a task cannot be declared as complete until the interpretation of all nets started within the task has been completed. Note that the call concept is recursive, i.e. process nets can be nested to any depth through the use of calls.

Task modules are special net modules which serve for the representation of tasks as components of process nets. They are characterized by the numbers and types of their ports: A task module has just one input port which is a trigger transition and a finite number of output ports which are sensor transitions. Moreover, it is required that the events associated with the event transitions must all be different. The semantics of all these ports is the following: At the moment when a task is activated - i.e. when the input port transition fires - the input trigger transition generates an event which signals to the corresponding actor[2] that s/he is being set this task. Having

1. The role of a person is the authorization allowing him/her to process a certain number of tasks.

completed the task - which can be specified by a complex refinement of the task module - the actor has to generate an event which corresponds to one of the events associated with the sensor output transitions of the task module.

An *elementary task module* is a task module characterized by a certain type of refinement and given the name *activity.* The refinement of an activity always consists of a trigger transition which forms the only input port, a finite set of sensor transitions which form the output ports and a single place which is connected by an incoming arc to the trigger transition and by one outgoing arc to each of the sensor transitions (cf. Fig. 11).

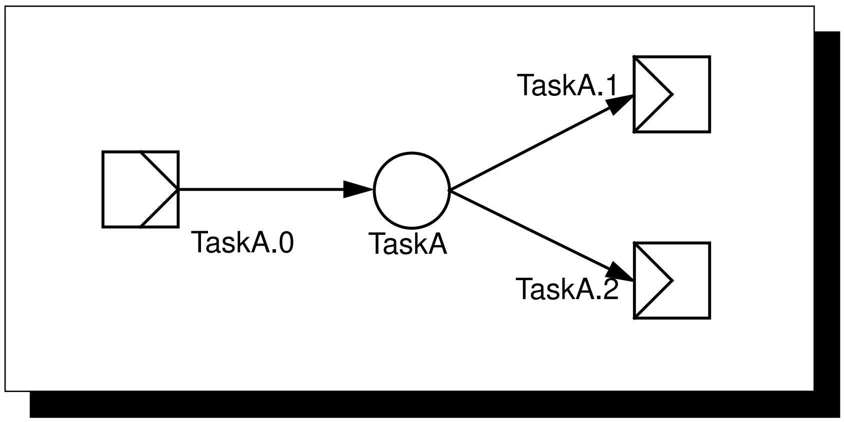

Fig. 11: Refinement of an example activity TaskA.[1]

Given an enacted process net, the event assigned to the trigger transition (here: TaskA.0) triggers the setting of the corresponding task to an actor whose role must be compatible with the role associated to the task. Conversely, the working environment of the person working on the task ensures that precisely one of the events which can trigger one of the sensor transitions (here: TaskA.1 or TaskA.2) is selected by this person, generally after a non-zero time interval within which the place of the net module is marked.

If the place is marked, then the activity is referred to as *active*, otherwise it is *passive*. The type of an activity is determined by the number of output ports and is therefore equal to a natural number. Activities of type 1 are called *ordinary activities*, those of types 2,3,... are called *decision activities*.

The refinement of an activity as shown in Fig. 11 using the example of a type 2 activity will not be shown on the level of process net visualization for user interaction. Instead, the symbols shown in Fig. 12 and Fig. 13 are used according to this special type of module refinement. In addition, ordinary activities can be characterized by graphical symbols referring to a certain application class of the task in question (e.g. “individual processing of a document” or “video conference”). In the case of decision activities, the drawn-in switch symbol can be treated as such a symbol (“decision activity”).

---

2. This actor may be a person or a piece of software.

1. In correspondence to the naming conventions given below, this activity is of type 2.

In the same way as the single input port is omitted, the output port of ordinary activities is not represented either. In the case of decision activities, the output ports are shown in addition to the drawn-in switch symbol (which makes clear the “either/or” relation between the ports) in order to visualize the “and” relationships between the arrows connected with one port.

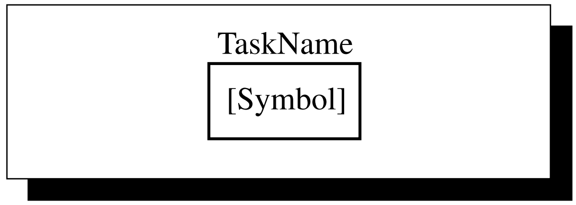

Fig. 12: Type 1 activity (*ordinary activity*)

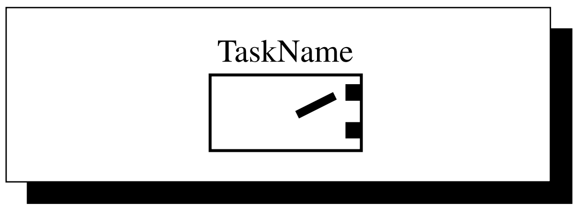

Fig. 13: Type 2 activity (*decision activity with two alternatives)*

Naming conventions

To enable compact but precise process descriptions which are as complete as possible, the names associated with the nodes of modular process nets are intended to play a useful role for the semantics of these nets. We do not require the following naming conventions to be part of the modular process net syntax, but they will be used in the examples in Section 7. Moreover, most of these conventions have proved very useful in experimental implementations and in the use of modular process nets.

First, consider the names of *task modules*. Here, both for elementary task modules (activities) as well as for complex task modules we allow (and recommend where possible) the syntax rule

TaskName:= Activity (Actor)

with the refinement

Actor:= Role | Person

The meaning of this rule is that the activity **Activity** which has to be done in order to perform the task is assigned to a concrete person **Person** or to a person with the role **Role**.

For an ordinary transition representing an *action* to be performed by a hardware or software device (but not by a person), a similar naming convention as for task modules can be applied, i.e.

TransitionName:= Action (Device)

Remember that it is recommended to use square boxes for elementary actions to be performed by hardware or software units and rectangular boxes for activities (elementary tasks) which are set to persons (with the fixed refinement as given in Fig. 11). Both notions can be used together in one process net as seen in the example “session management” below.

For a *decision activity* it is useful that **TaskName** is the question to be answered when processing the task (e.g. “Repair completed?”). The names of the postplaces of a decision activity should contain the possible answers to this question (e.g. “yes” and “no”). If there is just one postplace for a port, the name of the corresponding postplace should be *identical* to this answer. Thus, the corresponding syntax rule is

PlaceName:= “answer to the question **TaskName**”

In the general case, the names for places should represent names of *conditions* which may be true or false: If a condition is true at a certain moment in time, the place is marked, whereas it is not marked in the opposite case. Usually the condition states the availability of a certain piece or type of information. This information is “consumed” by firing of its posttransition[1]. As stated above, the conditions corresponding to postplaces of decision activities should contain the answers to the questions associated with the decision activities.

A usual case for the application of process nets is the representation of combined control and data flow in one net. In addition to the notification of the progress status of the control flow of a process, places may also represent

- *data* which is *required* for an action or for an activity and which will be consumed by them (this interpretation applies to the preplaces of a transition) and
- *data produced* by an action or an activity (this interpretation applies to the postplaces of a transition).

Note that every place of a process net arises in both roles - as preplace of a transition and also as one of the postplaces of a transition.

Marking of actions and activities

In the development and use of graphical tools for process nets it has proved useful to enable a graphical monitor indicating the current state of an enacted workflow to provide not only the current marking of a process net - i.e. the local states and data waiting to be processed - but also the currently active actions and activities and, moreover, information about completed processing. The latter information consists of the set of all actions and activities which have been active in the past and the conditions which have been true including the data whose processing has been completed. This “tracing information” can be - and has been - made available to the user by coloring the symbols representing the nodes of an enacted process net.

1. Due to the syntactical restrictions on process nets there can only be one posttransition

6.2 Process modules

Process modules are re-usable modular process nets which can be started (“enacted”) as administrative *procedures*[1] from within tasks by using their name and providing their parameters. Their use is supported by a process net library.

Unlike the elementary process nets defined in Section 5, process modules can also contain task modules in addition to transitions, as can be seen from the definition of modular process nets. On the other hand, the net structure of a process module - i.e. the underlying net - must conform to the five restrictions listed and explained below.

(1) A process module contains one and only one PN-start transition.

Explanation:
By triggering a PN-start event, the environment of a process net interpreter generates a new instance of the corresponding process net and at the same time its PN-start transition is fired by the interpreter. Because the case of several PN-start transitions - corresponding necessarily to the same PN-start event - in one net could be simulated by one PN-start transition with several output arcs, this case will be excluded for clarity of representation. On the other hand, without PN-start transitions in a net, the generation of new net instances would be impossible.
There is a certain similarity between sensor transitions and PN-start transitions: In both cases the net progress depends on the occurrence of external events. But, in contrast to PN-start transitions, the firing of sensor transitions is not connected with the generation of new net instances.

(2) A process module must contain at least one PN-end transition.

Explanation:
The firing of this transition causes the net instance to terminate and triggers the event associated with this transition in the surrounding net. In the case of several PN-end transitions, the information about the PN-end transition actually firing can be used to obtain information about the history of the finished process.

(3) No transition of a process net may be concurrently enabled with a PN-end transition.

Explanation:
This restriction is required to avoid ambiguities concerning the last activities taking place in a process net. Note that this consistency condition has to be verified by net analysis before storing the net in a process net library.

1. In German: “Verfahren”

(4) No two transitions of a process module (including the activities) have a common preplace.

Explanation:
This restriction ensures that no conflicts can occur on the coarse grain level of a process module. Note that on the refined level of a process net conflicts between the sensor transitions (corresponding to one task module on the coarser grain level) are possible. (cf. Fig. 11).

(5) The elementary process net resulting from the refinement of activities in a process net should always be safe, i.e. there is never more than one token on any of the places when using the normal enabling rules.

Explanation:
This property ensures that process modules describe “reasonable” behavior. In particular, the state of a process described by a process module is uniquely characterized by the set of valid conditions, i.e. by the set of marked places. Analysis techniques developed in the literature (e.g. [Star90]) and implemented in corresponding tools (e.g. [Star94]) can be used to obtain the safety property for a given net.

An *elementary process module* is defined as a process module which does not contain any complex task module, i.e. each of its task modules is an activity. The advantage of this notion is completeness of description: Not only the refinement to an elementary process net is unique - due to the uniqueness of the refinement of an activity - but also the status information, since any activity has one and only one place, implying only two possible markings of the subnet.

For better comprehension let us finally illustrate the relationships between the notions introduced in the previous chapters in the following diagram:

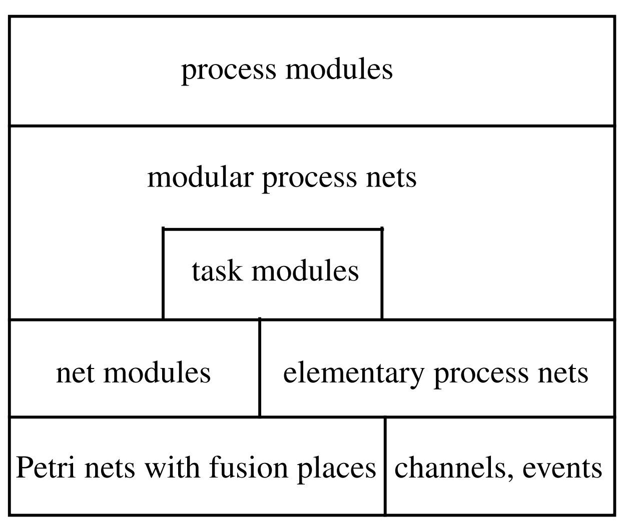

Fig. 14: Relationships between the introduced concepts

7 Examples

This chapter presents some simple but typical examples of process nets taken from the areas of workflow modeling and software architecture procedure description.

The example in Section 7.1 is a typical part of a business process arising in enterprises which have to prepare offers at the request of clients. Moreover, this process is designed to be enacted by a workflow management system, so that the model has to be understood as a prescriptive one.

The other example, outlined in Section 7.2, describes that part of the negotiation process for quality of service (QoS) which is realized by a software architecture called CMA currently being developed by the network group of the International Computer Science Institute at Berkeley. This model is intended as descriptive and aims at a better understanding and possible correction of processes to be programmed and at validation and verification of this software.

The examples were edited using the graphical tool MoPEd (**Mo**dular **P**rocess net **Ed**itor) which in turn generates a data structure which can be used as a basis for computer-based interpretation, storage and transformation of modular process nets.

7.1 Workflow model: Prepare offer

The process modules shown in Fig. 15 (“PrepareOffer”) and Fig. 16 (“AppointmentMaking”) are enactable parts of typical business processes where the refinements of the task modules (activities) are assumed to be known from Fig. 11 to Fig. 13.

The aim of the process to be controlled by the process module “PrepareOffer” (see Fig. 15) is the preparation of an offer for an external client by a group of specialists. The process is initiated and supervised by a person with the role “group leader” (for short GrL) and carried out in parallel by at least two specialists M1 and M2 (“members of staff”) who process the business and technical parts of this task more or less independently. A typical part of the technical processing is an optional meeting to be held with other specialists. Therefore, the decision whether to hold the meeting or not is modeled as a particular activity of the technician M2. Having received the results from M1 and M2, the group leader completes the offer, thereby completing the process prescribed by the process module “PrepareOffer”. Note that this process module was started by the group leader for better structuring and control of his own work, but several tasks are delegated to his colleagues M1 and M2 to whom he is entitled to set tasks. As can easily be seen, the process module contains “and” branches and confluences of the control flow[1] as well as “or” branches and confluences[2].

1. *The “and” branch begins after the task “Delegate partial tasks (GrL)” and rejoins with the task “Complete offer (GrL)”. The start of this last activity usually involves waiting for synchronization of both threads of control*

2. *These are connected with the decision tasks “Technical meeting necessary?” and “Find date for meeting” (“or” branches) and the place “partial result technical” (“or” confluence)*

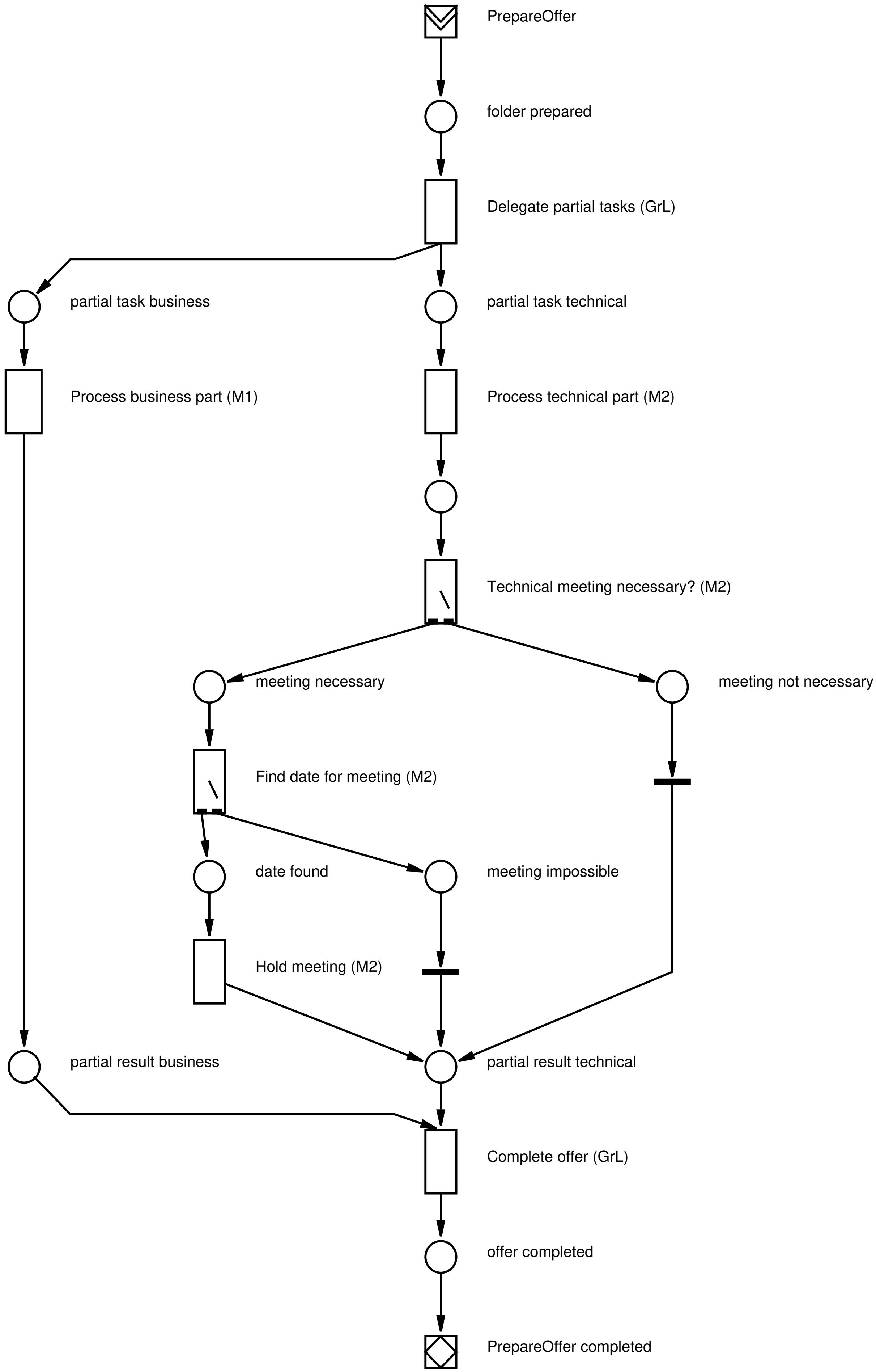

Fig. 15: Process module “PrepareOffer”

The process module “AppointmentMaking” (see Fig. 16) can be used as technical support for M1´s task “Find date for meeting” in the process module “PrepareOffer” in Fig. 15. It can be started by M1 if s/he finds it useful to do so, but can also - with a slight modification of the process module “PrepareOffer” - be started automatically by this process module. Note that no roles or persons are assigned to its tasks because it is assumed that all tasks are set to the starter of the process module.

After the completion of the first task by M1, i.e. after sending the date proposals to all parties who are to participate in the meeting, the process net resides in a state where the conditions “waiting for deadline” and “wait” are true. Remember that the delay-free set-alarm transition initiates an event of the type EventName with a specified delay after its own firing. During this time, responses can arrive at the channel “new response to date proposal” and be included in the database by the activity “Include new responses”, resulting every time in the marking of the place “new responses included” which is followed immediately by the setting of the task corresponding to the decision activity “All parties responded?”. This activity is performed each time a new set of responses has been included and may eventually result in the condition “all parties responded”.

If this condition is reached before the occurrence of the event “deadline” (and thus before the firing of the corresponding sensor transition), this event will be cancelled by the clear-alarm transition and therefore, the place “waiting for deadline” will remain marked until the end of the interpretation of the net. The next activity will be the decision “Do ideal dates exist?” which checks for the existence of a “perfect set of dates”, i.e. such sets of dates where all invited parties are able to participate. If such a set exists, one of the dates will be chosen by the activity “Choose date” and the last activity of the net ensures that all parties are informed.

If, in contrast, the event “deadline” occurs before all parties have responded, the corresponding sensor transition will fire and consume the tokens from the places “wait” and “waiting for deadline”. Thus, even in the case that all parties responded - but after the occurrence of the deadline event - the clear-alarm transition will not fire and a token will remain on the place “all parties responded” until the end of the interpretation of the net.[1] Then, the task “Evaluate incomplete set of responses” will be set.

After this activity or if there was no date suitable for everybody (as a result of the decision activity “Do ideal dates exist?”), the process module will enter (and pass) the condition “imperfect set of dates”. This condition is followed by the setting of the task “Set date or cancel meeting” which in turn is followed by the activity ensuring the notification of all parties.

1. *If not all parties responded, a token will remain on the place “incomplete set of responses” until the end of the net’s interpretation.*

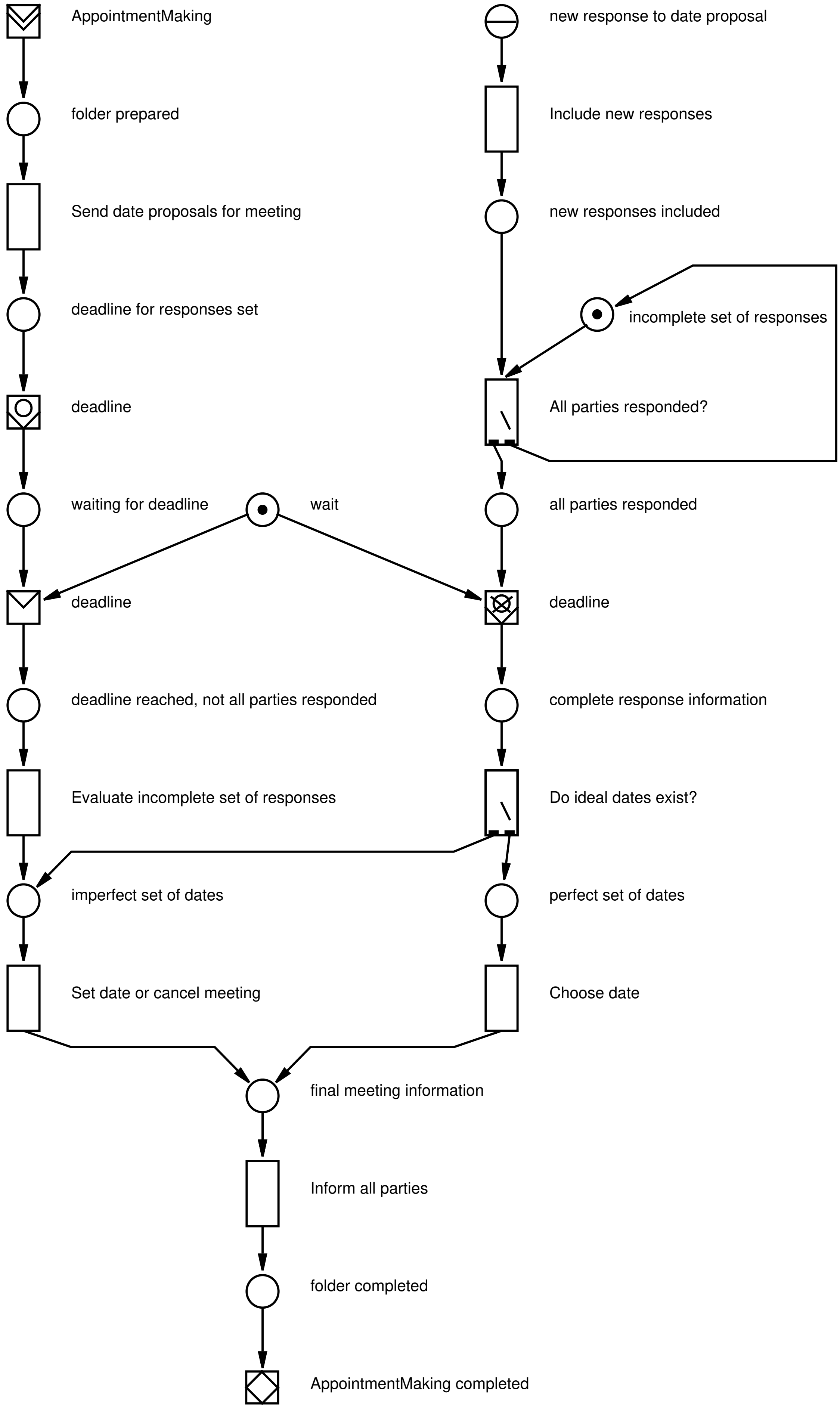

Fig. 16: (“AppointmentMaking”)

7.2 Processes in a software architecture: Negotiation for quality of service

A topical area of research of the network group of the International Computer Science Institute at Berkeley is the development of methods and software architectures which enable (partly statistical) guaranties for the quality of service (QoS) provided by broadband end-to-end communication services on the host level. To this end, a so-called CMA (Cooperative Multimedia Application) architecture is being developed as part of a host's software architecture.

An important part of the functionality of this architecture is the negotiation process for the quality of service. This process is realized by interaction of the sub-components of the host's major component Session Manger called user interface, QoS mapper, service manager, resource monitor/controller and connection manager.

Usual graphical representations in the design of software architectures reflect only the basic functionality and the (names of) services provided by the software components, but not the complex processes which are realized by their interaction (see Fig. 17).

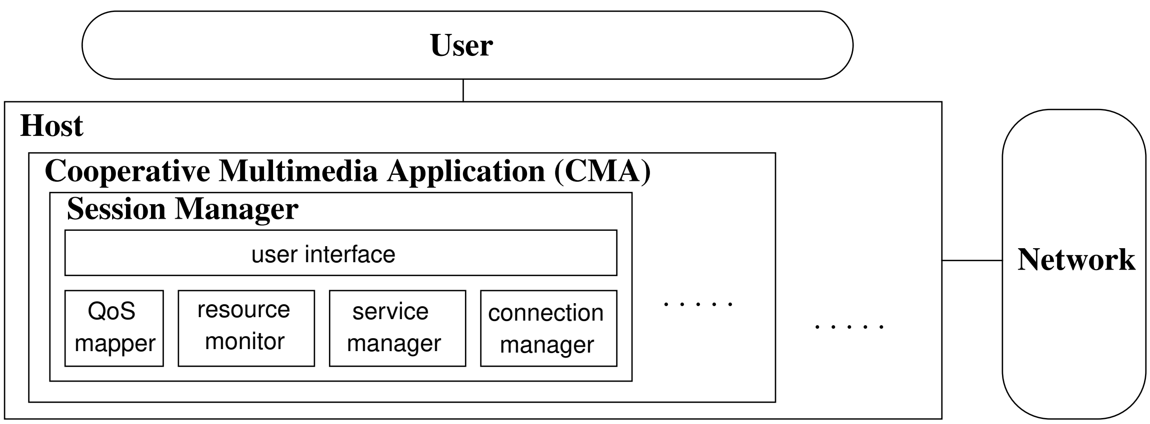

Fig. 17: CMA software architecture

Modular process nets are well suited for a detailed and formally correct graphical representation of such processes. They can simultaneously serve for easier understanding, documentation, verification and validation of the corresponding software design and implementation. A further (potential) use is the design of a user interface.

In our following (hopefully) self-explanatory example given by Figures 18 to 20, modular process nets are used for descriptive modeling - there is no intention to enact the specified processes. But, in contrast to the workflow example, the refinement possibilities of modular process nets are extensively used. Note that most of the transitions are *actions*[1], i.e. they take time to be completed and they can be imagined to be "marked". But, in contrast to activities, for actions no predefined refinement is given.

1. *The only exceptions are the user activities "Specify session requirements" and "Decide on acceptance of negotiated QoS"*

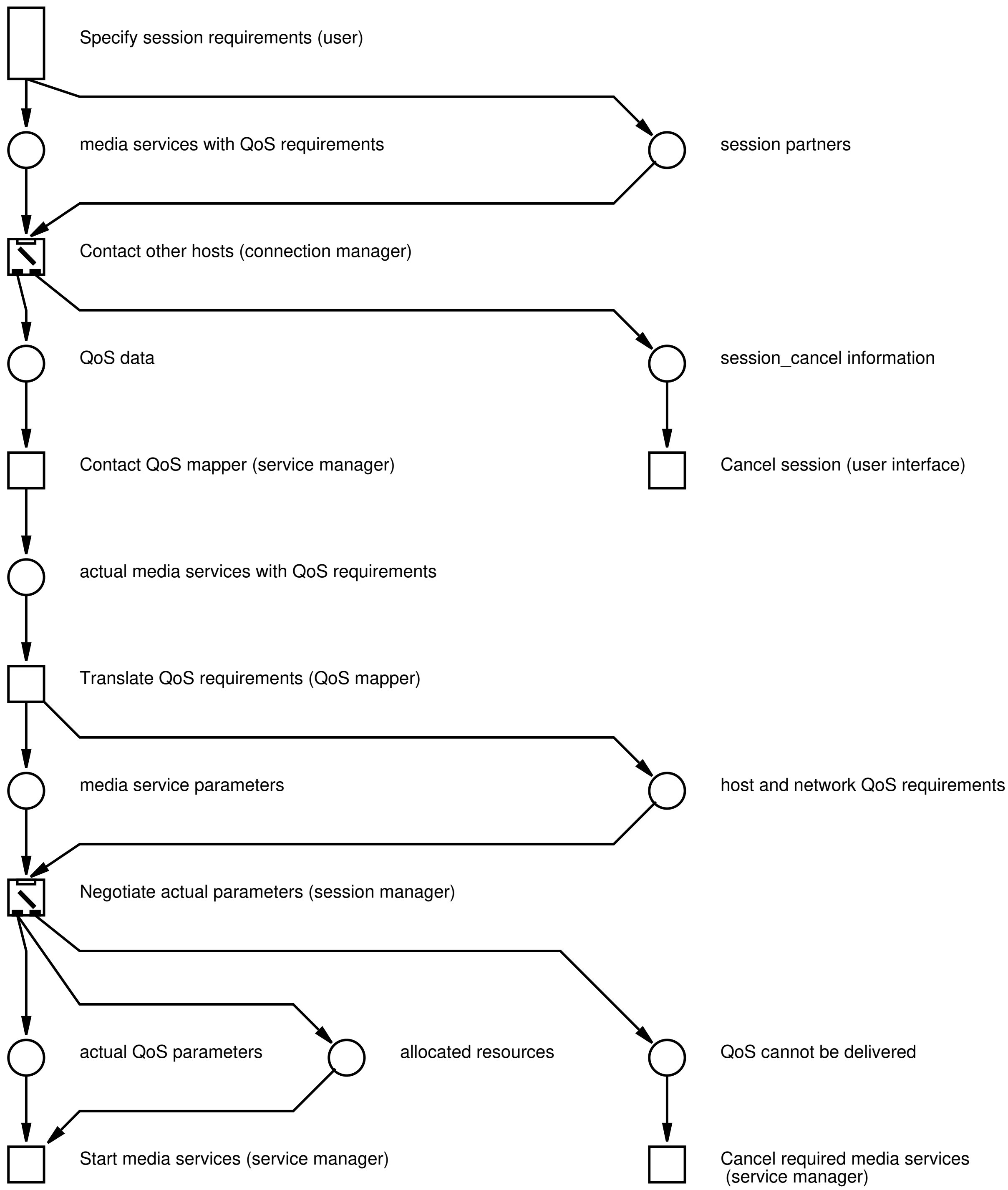

Fig. 18: MPN model “Session management”

”

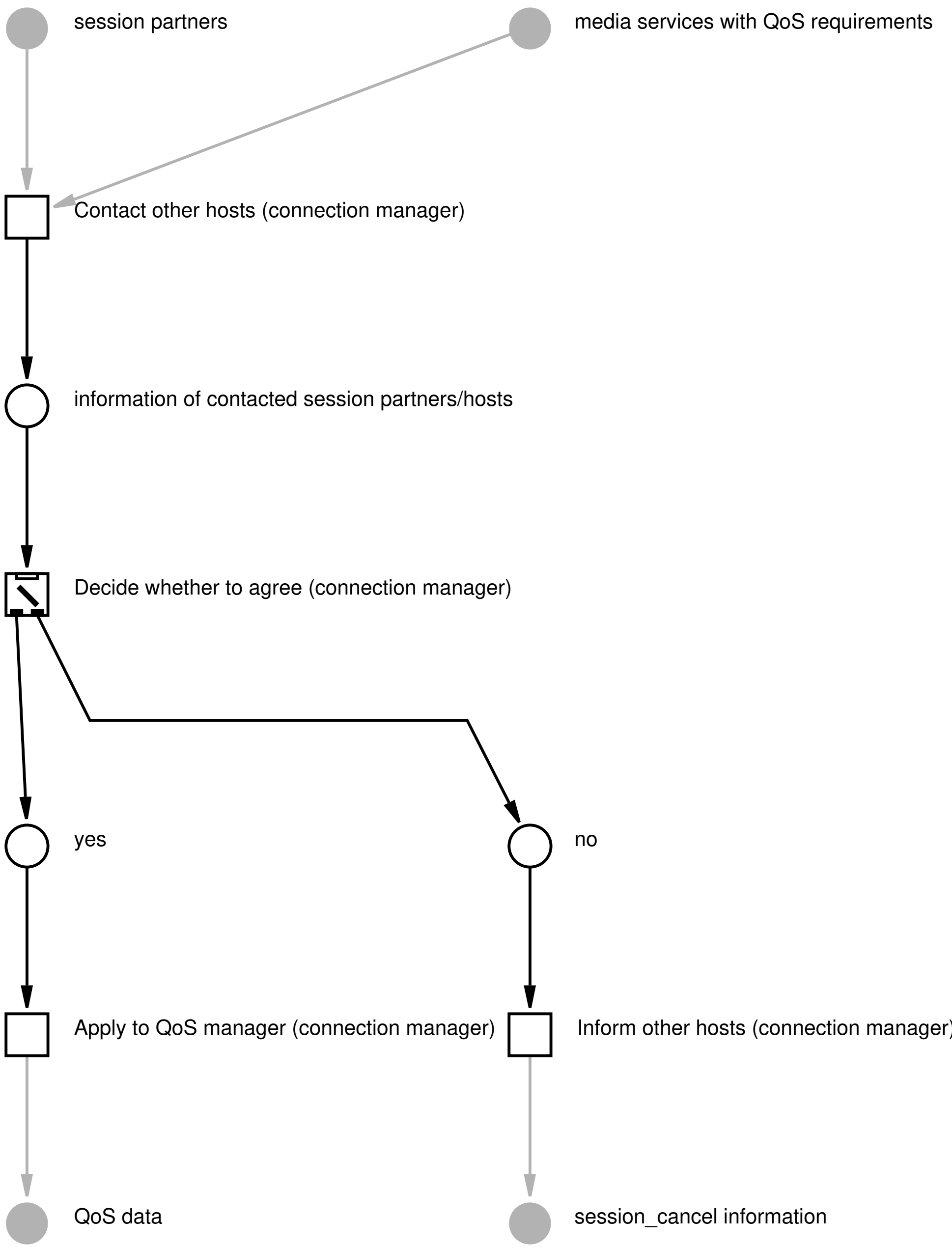

Fig. 19: Refinement of “Contact other hosts (connection manager)”

”

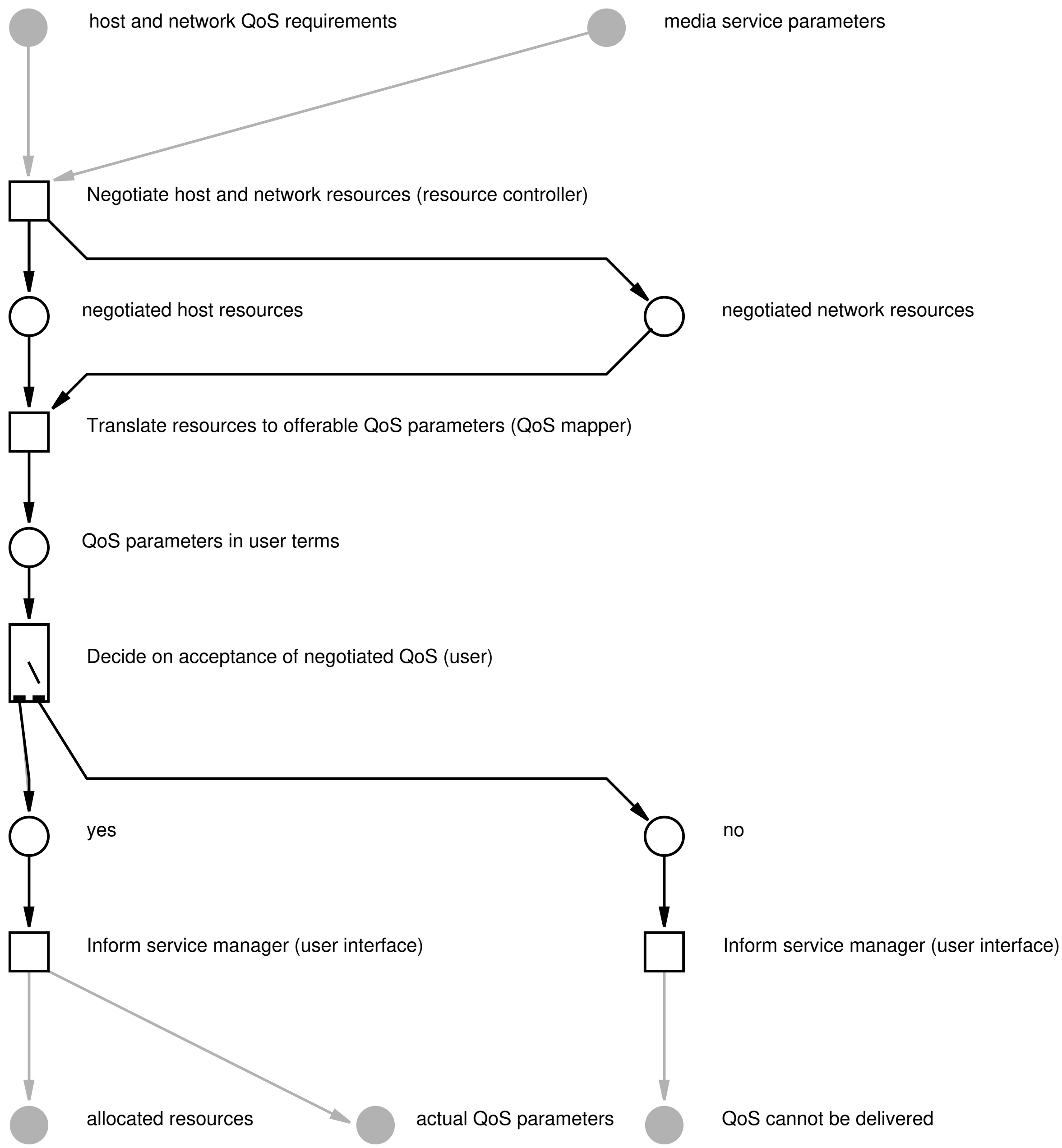

Fig. 20: Refinement of “Negotiate actual parameters (session manager)”

8 Concluding remarks

This report provides an introduction to a new class of Petri net models which is particularly well suited as a graphical and formal notation for business process and software process models.

The main and most innovative points of this class of models are its hierarchical module concept, the constructs for synchronous and asynchronous communication between interpreted nets and their environment and the use of these concepts as a basis for a canonical framework for flexible workflow modeling and enactment.

The module concept is part of a more general ("object-based") approach to Petri nets which not only allows a compact representation of complex processes but also provides solutions for problems in the application and use of Petri nets which result from the application of different firing rules and the introduction of time into nets.

In principle, the definition of constructs for synchronous and asynchronous communication can be done independently of the module concept for every class of Petri nets. The necessary underlying concepts for these constructs are those of events (memoryless "flashes") and of token passing (comparable to messages).

One very useful application of the module and event concepts is the construction of a net-based task concept which has been successfully applied to the design and use of a workflow management system but which is also expected to be applicable in other areas of computer-supported cooperative work (CSCW).

One of the main aims of the proposed class of models is that it should be simple, easily learnable and comprehensible in order to be used as a widespread but formally precise means of communication. For this reason in particular, only "black" tokens (instead of the often used class-based or individual tokens) are allowed in our modular process nets.

The other design objectives - to allow compact representation of complex processes, automated computer interpretation, distributed and safe interpretation and enactment, changes of the net structure during interpretation[1] and integration with organization models - have been achieved by minimal syntactic extensions of elementary low-level Petri nets.

Within the context of the WAM[2] project at the Fraunhofer ISST Berlin [Adam95], a set of several software components has been developed or enhanced which allow prototypical use of modular process nets for the modeling and enactment of flexible workflow processes. In addition to the graphical editor MoPEd - used for the construction of all process nets shown in this report - these are a process net interpreter and a process net monitor which have been integrated in a CORBA-based workflow management environment. Using a formal translation, the process net models can be analyzed using the Petri net analyzer INA [Star94].

As shown in the second example of this report, modular process nets are well suited for use outside workflow modeling and management. It is expected that applications in many other areas will follow. The main aim of this report is to encourage and promote such further use.

1. *This can be realized using the task concept and embedding of process modules.*

2. *Wide Area Multimedia group interaction*

Acknowledgments

The development of the first (previous) version of modular process nets has much profited from discussions with the members of the WAM project at Fraunhofer ISST in Berlin, in particular with Gert Faustmann and Burkhard Messer.

For the excellent working conditions while writing most of this report and for the provision of the QoS application example I have to thank the ICSI at Berkeley and in particular the members of the network group.

Last but not least, I would like to thank Nick Grindell for translating parts of the report and for his critical reading which led to a thorough revision of many of the English formulations.

Of course, I alone am responsible for any kind of errors.

References

[Adam95] Adametz, Helmut; Barthel, Beate; Faustmann, Gert; Messer, Burkhard; Wikarski, Dietmar; Fellien, Arne:
The WAM Project. Final Report (in German),
Berlin, Fraunhofer ISST, Internal Report 5/95, September 1995

[Baum90] Baumgarten, Bernd:
Petri-Netze - Grundlagen und Anwendungen (in German),
Mannheim, Wien, Zürich: BI-Wissenschafts-Verlag, 1990

[Ciar95] Ciardo, Gianfranco:
Discrete-time Markovian Stochastic Petri Nets,
in: Stewart, W. J. (Ed.) Numerical Solution of Markov Chains '95, Raleigh, NC, 1995, pp. 339-358.

[HaWi95] Hahmann, Andre; Wikarski, Dietmar:
MoPEd (Modular Process net Editor) Version 3.2
Berlin, Fraunhofer ISST, 1995

[HaWi95a] Hahmann, Andre; Wikarski, Dietmar:
Petri net tools - a comparative study (in German)
Berlin, Fraunhofer ISST, Internal Report 6/95, December 1995

[Hein92] Heiner, Monika:
Petri Net Based Software Validation - Prospects and Limitations
Berkeley, International Computer Science Institute, TR-92-022, March 1992

[HeVW94] Heiner, Monika, Ventre Giorgio, Wikarski, Dietmar:
A Petri Net Based Methodology to Integrate Qualitative and Quantitative Analysis,
Information and Software Technology 36 (7) 1994, pp. 435-41.

[LoWH95] Loewe, Michael, Wikarski, Dietmar, Han, Yanbo:
Higher-Order Object Nets and Their Application to Workflow Modeling,
Technical University Berlin, FB Informatik, Forschungsberichte 95-34

[Reis85] Reisig, Wolfgang:
Petri Nets - An Introduction.
Berlin; u.a.: Springer, 1985
(Springer Compass)

[Reis95] Reisig, Wolfgang:
Progress in Petri Nets.
Manuscript, Humboldt-University Berlin, Fachbereich Informatik, 1995

[Star90] Starke, Peter H.:
Analysis of Petri Net Models (in German),
Stuttgart: Teubner, 1990

[Star94] Starke, Peter H.:
INA (Integrated Net Analyzer) Version 1.4
Berlin, Humboldt University, 1994

[Whit93] Whitworth, Jenny:
The Structure of Object Nets.
Stafford, UK, School of Computing, Staffordshire University, February 1993

[Wika90] Wikarski, Dietmar:
Object Nets - a Canonical Class of Models for Behaviour Simulation and Structure Synthesis of Distributed Systems?", Modelling, Evaluation and Optimization of Dependable Computer Systems, Proc. Intl. Seminar 1990, iir 6(1990)12.

[Wika95] Wikarski, Dietmar:
Modular Process Nets. User Handbook (in German),
Berlin, Fraunhofer ISST, Project report, February 1995

[WiHe95] Wikarski, Dietmar, Heiner, Monika:
On the Application of Markovian Object Nets to Integrated Qualitative and Quantitative Software Analysis
Berlin, Fraunhofer ISST, October 1995
(ISST-Berichte 29/95, ISSN 0943-1624)

Index

A

C

D

E

F

I

L

M

N

O

P

R

S

T

U