On the Application of Markovian Object Nets to Integrated Qualitative and Quantitative Software Analysis

Dietmar Wikarski

Fraunhofer Institute for Software Engineering and Systems Engineering
Kurstraße 33
D-10117 Berlin
Germany
E-mail: dwikar@isst.fhg.de
Phone: (+ 49 - 30) 20 224 - 847
Fax: (+ 49 - 30) 20 224 - 799

Monika Heiner

Technical University of Cottbus
Department of Computer Science
Postbox 101344
D-03013 Cottbus
Germany
E-mail: mh@informatik.tu-cottbus.de
Phone: (+ 49 - 355) 69 - 2794
Fax: (+ 49 - 355) 69 - 2794

Abstract:

In this paper a Petri net based methodology is outlined for an integrated qualitative and quantitative analysis of parallel software systems which is based on different (Petri) net representations of the software system under consideration.
The software validation methodology starts from the source text of a set of communicating processes which specify the system under development. From this source text skeleton, a Petri net representation of the general control structure of the system is generated. On this basis a set of reduction steps is defined for transforming the net into different intermediate representations allowing the validation of both qualitative and qualitative properties. Particular attention is paid to the validation of quantitative properties which is performed using a transformation into locally Markovian Object Nets (MONs). Models of this new class of modular Petri nets are obtained by property-preserving structural compression and by enhancement with quantitative information. This information is obtained in the form of frequency and delay parameters by monitoring and testing the software on the basis of prototypes.
Compared to a previous paper on the same topic /Heiner94/, the methodology as well as its presentation have been improved. In particular, a new, more comprehensive running example has been selected and the object net approach is explained in much more detail.

Keywords:
Performance models, Software validation techniques, Parallel systems software engineering, Formal methods, Petri nets, Object nets.

1 Introduction

Parallel programs are inherently concurrent and asynchronous. Their behaviour often depends critically on the potentially unpredictable timing of their components. The large numbers of subtle interactions that can take place among the components of even a moderately-sized parallel system make it extremely difficult to evaluate the dependability properties of the system's behaviour.

Therefore, powerful techniques are needed for rigorously analysing the possible kinds of behaviour of parallel software systems to assure that they exhibit all and at best only the properties intended. Because of the complexity of the arguments involved, developers of dependable parallel systems would benefit greatly from automated tools to aid in the analysis of the system they are going to create.

Software Validation tries to minimize the presence of faults in the operation phase by analytical and (as far as possible) computer-aided methods in the pre-operation phase. Because different validation methods aim generally at different properties, qualitative as well as quantitative ones, it is obvious that no single one is able to guarantee complete confidence in the desired total software quality. All approaches have their advantages and limitations. For this reason, they should not be viewed as competing, but rather as complementary techniques.

Model-based validation approaches have in common that they rely on some **formal model** of the actual software. These models usually reflect only certain aspects of the modelled objects in the hope that the abstraction will lead to problems of manageable sizes and that the models will cover a range of viewpoints forming a sufficiently complete view of the software. In other words, the models neglect all those properties of the software to be analysed which are supposed not to be important for the properties under consideration.

A more detailed differentiation of the model-based methods relies on the different software properties in question:

- **Context checking** deals with general qualitative properties like freedom from data or control flow anomalies which must be valid in any system independent of its special semantics (for that reason, it is sometimes called general verification). These properties are generally accepted or project-oriented consistency conditions of the static semantics of any program structure.
- **Verification** aims at special qualitative properties like functionality or robustness which are determined by the intended special semantics of the system under development (to underline this fact, it is sometimes called special verification).
- **Evaluation** techniques treat quantitative properties like performance and reliability to predict the software's timing behaviour in advance, or to assess it afterwards.

While the properties the evaluation deals with are inherently time-based, both context checking and verification aim at time-less properties which should be valid independently of time. Unfortunately, that is not always true in the case of parallel programs (see section 4).

It is well documented that debugging of parallel programs only by means of systematical run-time tests is hardly possible. This is because some faulty behaviour such as synchronization errors, like total or partial system deadlocks, can be controlled by the current progress of the processes, which is generally time-dependent and non-reproducible.

Moreover, in /Gait 86/ the experience is reported that some synchronization failures could not be observed by debugging (without hardware support). This phenomenon of error masking which is called **probe effect** and which can be interpreted well in terms of Petri nets, emphasizes the need for a means of software quality assurance other than testing.

It is an (at least implicitly) common opinion in software development technology that the relative progress of processes with respect to each other must not have any influence on the general logical behaviour of the system as a whole.

2 Overview of the Net-Based Methodology

Qualitative as well as quantitative properties involve interactions among the parts of a parallel system. It is quite widely accepted that these properties are most naturally analysed in terms of the order (and number) of events. Especially for these problems, Petri nets offer a suitable mathematical background with an already powerful theory which is still under development.

The Petri net based software validation approach is able to combine the advantages of high-level (specification/programming) languages with those of Petri net theory. Figure 1 shows the logical architecture of the proposed tool kit for net-based software validation with its components and their interconnections. The tool kit as a collection of different tools reflects the belief that a thorough software validation can only be reached by a suitable combination of different techniques which complement each other because each of them aims at different properties.

An important property of the Petri net approach is its extreme generality. It aids developers in a general way in reasoning about the behaviour of parallel systems. Because of its generality, the Petri net framework can be used with parallel systems expressed in a wide variety of specification or programming languages.

The semantics of a particular language is captured by a procedure for automatically deriving Petri net representations for any parallel system expressed in this high-level language. The modelling of parallel systems by Petri nets resulting in an intermediate representation yields several advantages.

First, tools for analyzing and reduction extract information about events and their ordering directly from this intermediate representation of the system. Therefore, they do not rely on any special assumptions, but concern general features of parallel systems and can be applied to any system once a Petri net representation for the system has been obtained.

Second, due to the generality of the Petri net approach, tools based on Petri nets can be extended to provide common analysis methods across a number of phases in the software development process as soon as some formalized description of the parallel system under development is available. This might be a source of valuable commonality and integration in a software development environment.

Third, the net-based intermediate program representation serves as a common root from which different net-based software validation methods, basically on communication/synchronization level, are able to start, e.g.

- analysis of qualitative properties like context checking of static semantics and verification of functional behaviour (see section 3),
- monitoring and analysis of quantitative properties (see section 4 onwards).

Here, net-based *monitoring* aims at providing quantitative parameters like time (delay) and frequency attributes which have to be added to the qualitative model in order to form a quantitative net model. The appropriate control points in the software being tested at which to establish such counting monitors can be deduced from the quantitative net model because it determines especially the suitable time abstraction level.

These different validation methods may require net models which vary partly in their level of abstraction (e.g. granularity of considered control/data flow, quantitative information). But all transformations (from the parallel software system description into a first qualitative Petri net model, and between the different kinds of net models) can be done formally, and therefore automated to a high degree. This includes especially the innovative approach of an as far as possible formal transformation of the qualitative model into a quantitative one.

The automation of all transformations also strengthens confidence in the concrete models for several reasons. First of all, the user is relieved of unnecessary details. Furthermore, the situation is avoided where the model becomes better than the original program source text (as happened, for example, in /Balbo 92/).

3 Net-based Qualitative Analysis

3.1 Introduction

Net-based qualitative analysis has been a well-known approach for about fifteen years /Heiner 80/. But putting it into practice is still a promising challenge because of the analysis algorithms' complexity and the fact that the results gained in the analysis phase are very sensitive to the abstractions done in the modelling phase. A related tool kit consists basically of the four components Petri net generator, linker, analyser and the error reporting and interpretation system /Heiner 92/.

The Petri Net Generators in use (PNG_{PDL} /König 85/, /Grzegorek 91/ and PNG_{C} /Czichy 92/) produce (place/transition) Petri net[1] representations of the (fine-grained/coarse-grained) control structure of a given sequential process. Basically, this generation

1. In this paper, only new technical Petri net terms are introduced formally. For related definitions of general Petri net notions see e.g. /Starke 90/.

assembles general Petri net components (see Figure 2) for all relevant linguistic means according to the syntax tree. Besides this basic functionality, further information is supplied which allows

- an automatic layout of the generated net afterwards, and
- an assessment of the program's structural complexity (Number of Acyclic Paths (NAP), see /Heiner 88/).

After that, the graphical Petri net EDitor PED /Czichy 93/ is used for visualization and linking of the generated Petri net parts, and for adding any supplements that might be required, e.g.

- modelling of all control variables and related operations, upgrading the generated control structure model to a control flow one (the automatization of this step is in preparation),
- modelling of the system environment behaviour,
- fault models (e.g. message loss or duplication induced by erroneous message channels).

Finally, the filtering capabilities of PED output the particular data structures required by the analysis tools in use (INA /Starke 92/, PROD /Halme 95/).

The validation of qualitative properties comprises two steps. First, the context checking of general semantic properties is done by a suitable combination of static and dynamic analysis techniques from Petri net theory. Afterwards, the verification of well-defined special semantic properties given by a separate specification of the required functionality is performed. During this second step, the power of classical Petri net theory is supplemented by the model checking approach, using temporal logic as a flexible query language for asking questions about the (complete/reduced) set of reachable states.

All analysis tools used have to be understood as implementations of well-proved theorems of Petri net theory. As a consequence, the results obtained by these tools are expressed in terms of Petri net theory, independent of any underlying special semantics of the software under consideration.

As an advantage, Petri net theory offers different methods for analyzing qualitative aspects of parallel systems, e.g.

- animation by playing the token game,
- static analyses by net reduction, structural analysis or net invariant analysis,
- dynamic analyses by complete/reduced construction of the system's state space (reachability graph), possibly followed by
- model checking to evaluate the temporal relationship of logic formulae.

Net-based **animation** aims at functional behaviour simulation by playing the token game. The results gained depend on the abstraction level of the underlying net model. But in any case, net prototyping by token animation is only a confidence-building approach unable to replace exhaustive analysis methods.

All **static analysis** techniques have in common that they avoid the construction of the reachability graph. While reduction and structural analysis aim at context checking (of general semantic properties, which have to be fulfilled by any program whatever its special functionality), the invariant analysis corresponds mainly to verification (of special semantic properties, which are determined by the intended special functionality). By showing the existence of related net invariants, the net-based proof of program invariants shows some similarities to the classical approach of axiomatic program verification. First, suitable program invariants have to be hypothesized and second, the related net invariants have to be found from the (in general non-minimal) basis of invariants provided by a net analysis tool.

Dynamic analysis techniques have to be used if the static analysis efforts were not successful or if properties are wanted which cannot be analysed statically at all (e.g. freedom of dynamic conflicts, reversibility, firing of facts etc.).

In **model checking**, the reachability graph of a Petri net is interpreted as a database, and temporal logic is used as query language for asking questions about it. Therefore, all properties which can be expressed in the used version of temporal logic (CTL - Computation Tree Logic /Clarke 86/) can be checked. In this way, even very large reachability graphs become manageable.

Some qualitative software properties to be validated may require the use of higher-level net classes (at least coloured nets as short-hand notation of lengthy place/transition nets). The related models are developed by adding further information about the control data flow to the underlying control structures modelled by place transition nets. But we still try to restrict ourselves to those Petri net classes which support some type of exhaustive analysis (as opposed to simulation of the functional behaviour).

3.2 An Example

To make the basic ideas more clear, we will explain our approach using the concise example of (asymmetric) *competing servers*: There are two (operating system‘s) processes providing special asynchronous **services** (like printing) to the environment, represented here by a third process, the **requests** process. To keep the situation as simple as possible, let‘s suppose two services are available. The corresponding input queues of service requests are called chan1 and chan2. One of the servers, called **service12** process, is able to provide both services. The other server, called **service2**, is a dedicated process which is able to provide only one of the services, but with a higher quality. So the basic intention is that the server with greater functionality supports the dedicated server if the latter is no longer able to deal with all incoming requests.

In Figure 3 a sketch of a possible solution is given, written in ANSI C extended by parallel constructs ("send” and "receive” for asynchronous, indirect communication, "wait event” for non-deterministic waiting).

By means of this artificially constructed example, we will now demonstrate how to develop a model whose granularity depends on the validation method selected. In so doing, we will also be introducing different types of reduction procedures.

The automatically generated Petri net, modelling the fine-grained[1] control structure of theservice12 process is shown in Figure 4. To highlight the generation principle, the Petri net components assembled by the Petri net generator according to the given syntax structure are surrounded with shaded boxes. All node names are controlled by the appropriate Petri net components (see Figure 2), while the suffix numbers in the node names refer to source text line numbers. Transitions labelled with "do" model atomic sequential program parts without any inner communication actions. Thus, the consumption of time is their only influence on the system's behaviour.

Generally, the output of the generator can be optimized by reduction of those parts of the general Petri net components which are not used in the given context (e.g. unused break places of loop components, end branches of infinite loops). This is done during a first **weak reduction** step[2](in this case done by hand). The correspondence to the source text is fully preserved. The results which we obtain for our three processes are given in Figure 5 and Figure 6.

Please note the following drawing convention. Shaded nodes are so-called *fusion nodes*. They serve as connectors: all fusion nodes with the same name are physically identical. Thus, they will be merged when analysing control or data structures. Usually, communication objects are represented by such fusion nodes to avoid immoderate edge crossing.

The coarse structure given in Figure 7 provides an overview of the whole process system, where the corresponding fusion nodes (chan1, chan2) have been merged. It shows the top level of a hierarchically structured Petri net which we get as the result of the linking step. During linking, all (private) nodes of one process are uniquely prefixed to preserve node name uniqueness within the total system. Each of the macro transitions (represented as nested double boxes) includes the behaviour of one process on the next lower level (i.e. the net structures of Figure 5 and Figure 6, but with prefixed node names).

The basic core of the communication structure for qualitative analysis (communication skeleton for short) is given in Figure 8. We obtain this skeleton as the final result of a **firm reduction** procedure to abstract from all internal process behaviour with state machine structure which does not take part in the processes' interactions. The communication skeleton comprises mainly all those transitions which model communication language primitives. These communication transitions may be supplemented by further transitions which are required to form the right (coarse) control flow structure.

In the case of our process system, the two basic properties of general semantics (boundedness, liveness) can be decided very fast by **strong reduction**, whereby only the candidates for unboundedness, the communication places, are excepted from reduction. After several steps of applying general property-preserving reduction rules /Starke 92/ we get a net in the form of a sequential text description which can be visualized as shown in Figure 9. This net is obviously live and unbounded. To concentrate on the special topic of

1. If we are interested in qualitative analysis only, we would generate the coarse-grained control structure right away. Here, any sequential program parts which do not include process interaction primitives are reduced to one transition.

2. This optimization could also be done automatically by a more sophisticated Petri net generator, which is in preparation.

this paper, we will apply no further qualitative analysis methods to the running example at this point.

Figure 10 summarizes the order and interrelations of the reduction steps.

4 Net-based Quantitative Analysis: Overview

The basic idea of our approach to net-based integration of qualitative and quantitative analysis is to enhance an ordinary (qualitative) Petri net model with quantitative information in order to use the enhanced net with the same net structure also for the prediction of quantitative properties of the system under consideration. Here, quantitative properties are expressed by such system characteristics as delays by synchronization or transmission, throughputs of channels or processors, and probabilities for meeting deadlines or for being in some predefined state.

Such an approach is in contrast to conventional ones, which require separate modelling procedures to obtain such approved models as queueing networks or stochastic Petri nets.

Queueing networks: Besides the disadvantage of the additional effort to build a new model, this class of models imposes strong limitations on the system structure. More precisely, in terms of nets, the structure is required to be of free-choice type.

Generalized Stochastic Petri nets (GSPN), like the related class of **Deterministic and Stochastic Petri nets (DSPN**) /Marsan 87/, seem to fit well into the desired framework, but they have another disadvantage: in the general case neither liveness nor boundedness properties of the primary qualitative net model are preserved when considering the GSPN or DSPN with the same underlying Petri net. The deeper reason for this unfortunate property of GSPN and DSPN is the change of the firing rule. In a usual, untimed Petri net, the so-called "may (sometime)"- firing rule is applied: if a transition is enabled, it *may* fire, but doesn't have to do so. In GSPN and DSPN, as generally in the case of timed and stochastic nets, the "must (immediately after delay)"-firing rule has to be applied: any transition, when enabled, *must* fire immediately after a prescribed - possibly stochastic or equal-to-zero - amount of time ("delay"). Due to the "must"-firing rule[1], maximal sets of enabled transitions have to be fired simultaneously so that some transitions may never get the chance to become enabled.

As an example, consider the net depicted in Figure 8. In the untimed case, this net is live, but has the unbounded places chan1 and chan2. After changing to the "must"-firing rule the transition S12.receive.16 becomes a dead one and the places chan1 and chan2 become bounded. Notice that the change of the firing rule in this model has the same effect on its liveness and boundedness properties as a change to a discrete timed net (see, e.g. /Starke 90/), where all transitions are timed ones with the same constant delay.

Another problem of net-based analysis (both quantitative and qualitative) is the size of the nets in the case of more or less realistic programs. This requires the possibility of

1. For short, we will write in the following for "must (immediately after delay)"-firing rule, also in the case of a zero-delay, just *"must"-firing rule"* and for "may (sometime)"- firing rule just *"may-firing rule"*.

abstraction of subnets. Even in the case of constructing the net model manually, where such an abstraction is done by intuition, a concept of hierarchical abstraction and aggregation is desirable. But when the models are constructed automatically (as is the case in our approach) such a concept is unavoidable.

To solve these problems in an appropriate way, our approach uses a new, alternative concept of enhancing Petri nets with modularization and quantitative information. This concept is embodied in a new class of net models called **locally Markovian object nets** /Wikarski 92/, /Wikarski 94/, which combines the solution of both problems, i.e. the maintenance of qualitative properties also in the quantitative model and an appropriate abstraction concept. Additionally, it provides subnet abstraction by structured time parameters which may be used immediately for the quantitative evaluation of the system under examination.

5 Object Nets

The most general principles of the *object net approach* are

- the **modularization of a given net** in order to decompose it into *(net) modules* and
- an **abstraction of the module‘s behaviour** by *(net) objects* using some appropriate formalism, e.g. Markov chains /Wikarski92/, logical formulae /Whitworth 93/ or some other method (e.g. solution of conflicts by interaction with the environment of an executed net). In the case presented below, an abstraction by Markov chains is used together with a further abstraction by so-called defective discrete phase (DDP) distributions /Ciardo 95/.

In the following sections the concept and its applicability to net-based quantitative evaluation will be dealt with in more detail.

5.1 Canonical Modularization of Petri Nets: Net Modules

The common basic entity of modular nets and object nets is a new type of locality - *net modules* (and, after behavioural abstraction, *net objects,* see section 5.2) which is next greater than that of the usual type of locality in nets - transitions. *Transitions* describe *which elementary changes are possible* for given local states, i.e. markings of places, of a net. *Net modules* are those subnets of a Petri net which are explicitly equipped with interfaces and *represent connected*[1] *changes* (in some sense: local processes) of a given net. Net objects are net modules with behavioural abstractions (see below).

The guiding principle for declaring subnets of a net as its modules is the ability to locally solve *conflicts* which may arise between sets of concurrently *enabled* transitions of a module, i.e. without referring to the markings of other modules of the net. For the enabling of transitions there exist two different definitions. Below we consider only the more usual of the two which is sometimes called "weak enabling"[2]. A transition is ***enabled***[3], if all of

1. E.g. by a common observer or by a common interpreter, e.g. a processor with common memory.

its preplaces are marked. Based on this definition, two transitions are ***potentially in conflict*** if they share a common preplace.

Formally, a *net module* is defined as follows:

A subnet $N_M = [S_M, T_M, F_M]$ of an ordinary net N=(S,T, F) is called ***(net) module of N***, iff

1. For any transition t of the subnet N_M it holds that any transition of the net N which is potentially in conflict with t is also part of the subnet.
2. Any place **s** of N which may influence the enabling of a transition of N_M also belongs to this subnet.

Given an ordinary Petri net PN = [N, M_0], where N is a net and M_0 is the initial marking, the pair MN = $\{PN, N_{Mi}, i \in I\}$ with $N_{Mi} \cap N_{Mj} = \emptyset$, $N = \cup_i N_{Mi} \cup E_o$, $E_o \subseteq T \times S$ is called ***modular net***, iff the N_{Mi}, $i \in I$, are modules of N and form, together with E_o, a partition of N.

Any minimal module of a net is called ***basic module*** or ***basic object***[1]of this net. It can be shown that for a given net the set of basic modules is uniquely determined. Therefore, any module may also be seen as a composition of its basic modules.

Formally, the basic modules of a given net are defined as the sets of transitions obtained by the transitive hull operation to the potential conflict relation (these sets of transitions are also so-called *conflict clusters*) plus all preplaces of these transitions. For example, the basic modules in the net of Figure 13 consist of the sets of transitions {S2.receive.9, S12.receive.10, S12.receive.16}, {R.send.9}, {R.send.11}, {S12.end.wait.24} together with the respective preplaces and all arcs of the net which connect the places of a module with its transitions.

A rationale for this definition is the fact that the potential conflict relation is not transitive. For example, the pairs of transitions {S2.receive.9, S12.receive.16} and {S12.receive.10, S12.receive.16} are in a potential conflict, but the pair {S2.receive.9, S12.receive.10} is not. On the other hand, the firing of one of the transitions S2.receive.9, S12.receive.10 or S12.receive.16 influences the enabling of the two remaining transitions. Furthermore, the set of transitions remaining enabled may vary depending on which transition fires. Therefore, we require that every set of transitions formed by applying the transitive hull operation to the potential conflict relation of a given net must be completely contained in one module.

A similar proceeding, resulting in a different shape of modules, is possible in the case of "strong enabling" which for general Petri nets is characterized by limited capacities of places. In the case of a capacity of one for every place of a net, a transition is enabled if all of its preplaces are marked and none of its postplaces is marked. Therefore, two transitions are now in conflict if they share a common preplace or a common postplace. Thus, the postplaces of transitions must be taken into account when obtaining the set of potential conflicts.

---

2. The complementary case of "strong enabling" will be mentioned at the end of this paragraph.
3. In the case of ordinary Petri nets.
1. Both terms are equivalent, because any net object abstraction from a basic module (see section 5.2) has the same form as the considered basic module - in contrast to (general) net modules and net objects.

5.2 Net Modules with Behavioural Abstraction: Net Objects

Now, we will consider the behaviour of net modules in more detail in order to obtain behavioural abstractions of net modules which will be called *net objects.*
The local behaviour of a Petri net is determined by the enabling and firing rules of its transitions. The general local rule of the behaviour of a Petri net is: If a transition is *enabled*, it may or must[1] *fire*. In the case of ordinary nets as considered in this paper, ***firing*** means to simultaneously take one token from every preplace and to put one token on every postplace. In contrast to ordinary Petri nets with the "may" firing rule, the local firing rule to be considered in the following applies to the sets of possible steps of transitions of a module. In accordance with the intuitive meaning of an object (as well as with its meaning in object-based programming), modules with such a module-wide (= module-local = object-local) firing rule will be called (net) *objects.*

The *pragmatics* of net objects corresponding to net modules of a given Petri net is as follows. On one hand, when considering the net as a model derived from the *observed* behaviour of some system, the real-world entities corresponding to the places and transitions of the same net module are assumed to be *observable simultaneously.* In particular, the *simultaneous occurrence* of real-world events corresponding to different transitions of one net module is assumed to be observable - in contrast to the usual net pragmatics where this is considered to be impossible. On the other hand, when the net model is used prescriptively (e.g. when simulating the previously observed behaviour by the token game), the simultaneous firing of several transitions of one object is assumed to be possible as well. However, for transitions of different objects, simultaneous firing is assumed to be unobservable and therefore not allowed in a simulation.

Together with the guiding syntactical principle of putting transitions connected by the transitive hull of the conflict relation into one module, the *behaviour* of object nets is completely object-locally determined: all conflicts arising in an object net may be solved autonomously by object-local rules according to some internal algorithm which may be based on probability distributions obtained from the observation of corresponding frequencies in the real world. In our case, these probability distributions are assumed to be based on the monitoring of tests or other executions of the parallel software system under examination. Thus, in contrast to usual Petri nets, object nets allow a locally controlled automatic execution of a given net.

5.3 Locally Markovian Object Nets (MONs)

The basic idea of **locally Markovian object nets** *(MONs for short)* is the following: Based on a decomposition of a net into net modules and a subsequent qualitative behavioural abstraction of the net modules by net objects as described in the previous section, the assignment of marking-dependent probabilities to the possible steps (of transitions of these objects) is used for a quantitative abstraction of the behaviour of the objects and thus for an overall quantitative evaluation of the object net. Moreover, by applying the "must"-firing rule to certain sets of transitions, a further behavioural

1. The application of both "may" and "must" firing rules within one net will be considered in the next section.

abstraction is possible while preserving the qualitative properties of the net with the "may"-firing rule applied to all of its transitions. Let us consider this approach in more detail.

As stated in section 4, the use of the "must" firing rule in timed and stochastic Petri nets can lead to changes of the qualitative properties compared to the same net with the usual "may"-firing rule. For Markovian object nets we assume that both firing rules can be used for different transitions of one net, depending on the *type of the basic object* to which a transition belongs. If, for example, the "may"-firing rule is set for all transitions of the first basic object of the net in Figure 13 (S2.receive.9, S12.receive.10 and S12.receive.16)[1], while the "must"-firing rule applies for all the remaining transitions of this net, then the liveness and boundedness properties of this net are not altered compared to the same net with the "may"-firing rule applied to all transitions.

Generally, it can be shown that the firing rule of all transitions belonging to basic objects of free-choice type may be changed from "may" to "must" without effecting the liveness and boundedness properties of the net.

In order to use this property for an appropriate *qualitative behavioural abstraction* of net modules, the "must" and "may"-firing rules are assigned to its transitions in the following way: For all *input transitions* of a module - these are the transitions following the input places of a module - the "may"-firing rule is applied. All other transitions - they may be subdivided into *output* and *internal transitions* of the modules dependent on having input places as successors or not - are governed by the "must"-firing rule. Here, the internal behaviour of a module becomes characterized by a state machine where the states are the reachable markings and the state machine transitions are the possible ("module" or "object") *steps*, i.e. all possible sets of concurrently enabled transitions of the module. Based on local observations of the behaviour of the real-world entities corresponding to the state machines of the net objects, probabilities are assigned to the steps of transitions which are in conflict in the given object state. These probabilities determine the solution of conflicts between the possible steps. Note that this ***object-local probabilistic firing rule*** may restrict the previously applied "may"-firing rule by forcing some transitions to fire simultaneously (with a certain probability). If for each state with several poststates there is given a probability distribution over all poststates, the behaviour of the object is characterized by a Markov chain. A module with such a behavioural abstraction will be called Markovian object.

Object nets with the object-local probabilistic firing rule are called *locally Markovian* ones because their behaviour is on the one hand only locally determined (i.e. with respect to a single object) and on the other hand does not depend on past markings but rather only on the current marking. The latter property also characterizes (stochastic) Markovian processes, but with respect to a *global* state - an assumption which is dropped here.

Given the Markovian property, the probability distribution of the number of occurrences of internal steps from the firing of an entrance transition till the firing of an output transition may be taken as a characterization of the behaviour of the considered net module. So, an abstraction has been built which allows the former Petri net to be seen as a

1. This is the only basic object which is not of the free-choice type.

set of "weakly" (in view of the "may" firing rule between the objects) interacting objects called *locally Markovian object net*.

6 Application of MONs to Quantitative Analysis

The application of MONs to a quantitative evaluation of parallel programs while preserving qualitative properties is based on the combined use of the following findings.

1. If the "may"-firing rule is applied to all transitions of a MON - in this case the terms *net object, net module* and *basic object* are equivalent - and the probabilities for all possible steps of transitions are positive, then the liveness and boundedness properties of the MON do not change compared to those of the underlying net.
2. If the underlying Petri net is bounded, so is any MON based on this Petri net, regardless of which module partition is selected (and thus which transitions become governed by the "must"-firing rule). If, on the other hand, the underlying Petri net is unbounded, it is generally undecidable whether an object net remains unbounded by applying the "must"-firing rule to some of its transitions. For special cases, however, (un)boundedness may be decided.
3. If the "must"-firing rule is applied just to all transitions of modules with free-choice structure (in the following called ***fc-modules*** for short) of a modular net which is live when applying the "may"-firing rule to all of its transitions, this new modular net (modified according to the firing rule) will also be live.
4. Applying the "must"-firing rule to all transitions of fc-modules as described in Point 3. and interpreting all arising transition steps as transitions of a new net, this new net will have the structure of a state machine with the set of states equal to the set of reachable markings of the module. As a module with an abstraction of its behaviour is called object (cf. section 5.2 and section 5.3), this state machine will be called ***sm-object*** in the following.
5. Provided that (in the general case state dependent) probability distributions for the solution of conflicts are given and the probabilities for conflict solutions depend only on the current markings of the concerned objects, the behaviour of the (now: Markovian) sm-objects may be quantitatively characterized by DDP (defective discrete phase) distributions /Ciardo 94/.

6.1 Parameter Estimation and Simulation Experiments

Consider a Petri net which is decomposed into its non-fc basic objects (= non-fc basic modules) and arbitrary fc-modules and assume that for each reachable object marking a probability distribution over the possible steps of transitions is given. It can be shown that the resulting Markovian object net with the "must"-firing rule applied to the steps of transitions of the fc-modules and the "may"-firing rule to those of the non-fc basic objects has the same liveness and boundedness properties as the one with "may"-firing rule applied to all of its transitions.

This result can be used for the quantitative evaluation in the following way.

First, all necessary probabilities are obtained by counting the numbers of events corresponding to the conflicting steps of transitions by means of monitoring or by another type of estimation. For the fc-modules with the "must"-firing rule, these values are used as parameters of defective discrete phase (DDP) distributions as described in section 6.2.

Second, on the basis of the non-fc basic objects with the obtained marking-dependent probabilities and of DDP type time abstractions of the fc-modules, controlled simulation experiments are carried out in order to obtain those system parameters which ensure the boundedness, liveness and timeliness ("punctuality") of the system behaviour in a probabilistic sense.

Here, the simulation experiments may be carried out in a parallel or distributed way. The corresponding DDP delays may be sampled by a random number generator instead of the detailed simulation execution of the fc-modules. The "control parameters" of the simulation are delay and firing probability parameters of the non-fc basic objects which ensure that all local markings which are reachable in the untimed case will be reached in the simulation. For each of these markings, the firing is realized according to the object-local probabilistic firing rule.

As a result of the controlled simulation experiments we obtain conditional results depending on the firing delays and probabilities which can be used for an appropriate calibration/dimensioning of the system. Here, the introduction of local control or delay mechanisms into the real system may be necessary in some cases to ensure the required (quantitatively specified) boundedness and liveness properties.

As an example, consider once more our three party client/server system in Figure 8.

Using the known analytical results of Petri net theory, it cannot be decided whether this net is time-independently live (see /Starke 90/).

Assuming the "must"-firing rule and constant delays for all transitions, one can easily show that the previously live transition S12.receive.16 becomes a dead one when (1) the cycle time[1] of the service12 process is smaller than or equal to that of the requests process and (2) the cycle time of the service2 process is smaller than or equal to that of the service12 process. Due to (1), the service12 process is always already waiting for a request (i.e. place s12 is marked) when the requests process has generated a new chan1 request which is then immediately processed. Due to (2), the transition S2.receive.9 will permanently "steal" the token from the place chan2 before transition S12.receive.16 can get a chance to become enabled (because the service12 token is again in s12).

On the other hand, the weakly reduced net models of the service2 and service12 processes contain inner cycles whose duration may vary arbitrarily in dependence on the number of times the cycles are run through (see Figure 6 and Figure 6a). Therefore its cycle times cannot be modelled by the timed nets with constant delays as described above. When assigning probabilities to the transitions in conflict - the posttransitions of for.11 in Figure 6 and of for.11 in Figure 6a - the cycle times may show (with a positive probability) values of an arbitrary given size over and over again so that transition S12.receive.16 will

1. The time a token needs to cycle around one times.

be enabled at these times. Thus, the liveness of transition S12.receive.16 depends on "time structure" and parameters of the involved delays.

A solution of this problem provides the following approach: Using the measured delays and probabilities, those parameters are determined by controlled simulation experiments which ensure the required liveness, throughput, utilization and delay properties of the process system under consideration.

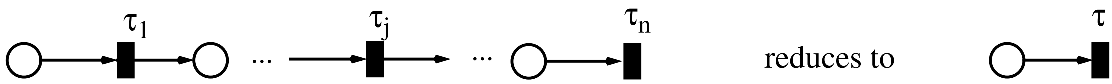
Which types of delay structures may arise in the fc-modules will be shown in more detail in the next section.

6.2 Quantitative Structural Abstraction by Delays with DDP-Distributions

The findings 1. to 5. stated above form the basis for a quantitatively evaluable abstraction from the internal structure of fc-modules. In particular, the (one-entry/one-exit) control structures of the sequential parts of parallel programs which have formed the focus of our attention are fc-modules (so far, even "state machine" ones) which can be assembled out of components of the types shown below. This may lead to a considerable reduction in the complexity of the resulting nets. The structural reduction of the mentioned control structures is based on the following rules which will be given together with its graphical net representations.

The basic abstraction (reduction) step is that of sequences of ***constant delay***s. A constant delay has a duration of an integer multiple of a ***basic time step***, which will be denoted in the following by Δ. Clearly, a composition of a finite number of sequentially ordered constant delays is again a constant delay. Graphically we will denote all types of constant delays, including the basic time step, by thick black bars. Moreover, for simplicity of notation we will denote all possible positive values of delays by τ_j. Note that $\tau_j = n_j\ \Delta$ always holds.

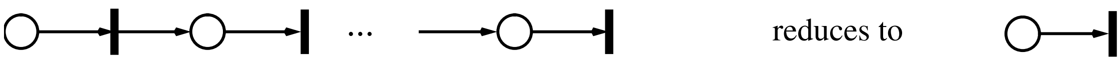
Constant delay (Const): Fixed number of sequential constant delays: $\tau = \Sigma_j\ \tau_j$



Here and in the following τ denotes the resulting (generally randomly distributed) delay of a time object. The symbol P(.) will denote "the probability of".

Immediate transitions are handled similarly: Two immediate transitions in sequence result again in an immediate one, whereas an immediate transition in a sequential composition with a non-zero delay transition (constant, geometric or DDP-distributed, see below) may be omitted.

Zero delay (Immediate transition): Sequence of immediate transitions (



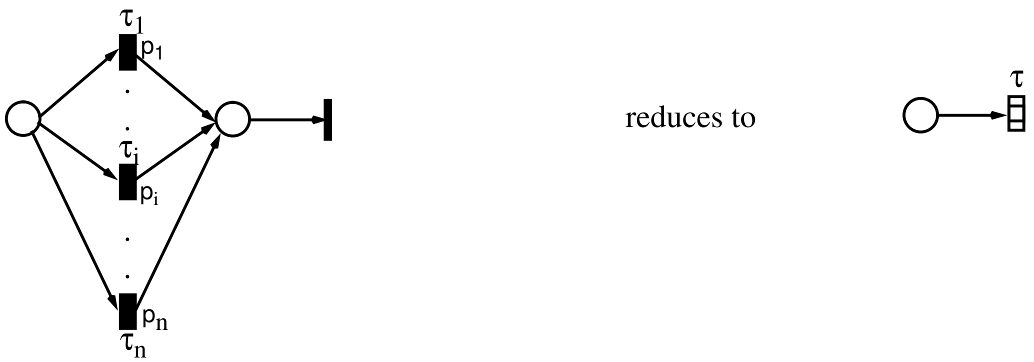
The elementary loop construct whose delay is geometrically distributed, and which is called ***geometric delay*** in shorthand notation, may be considered in the probabilistic sense as opposite to the constant delay. Note that the construction of a geometric delay requires two constant delays of identical duration. For a more convenient modelling of loop constructs we will therefore consider a ***modified geometric delay*** which ends with an immediate transition instead of with a constant delay whose length is identical to the inner loop delay.[1]

Modified geometric delay (ModGeo): Geometrically distributed number of identical constant phases decreased by one phase: $P(\tau = n\,\tau_0) = (1-p)\,p^n$, (n = 0, 1, 2,...)

The justification for using the same symbol for the geometric delay (in discrete-time models as considered here) as for the exponential delay (in continuous-time models) is their common memoryless property.

As the third basic type of delay we introduce delays with an arbitrary discrete probability mass function with finite support[2]. These delays may be represented as alternative compositions of a fixed number of constant delays:

Delay with an arbitrary probability mass function with finite support (= fixed number of alternatively chosen constant delays): $P(\tau = \tau_j) = p_j$, (j= 1,2,..., n)



Delays of all the types introduced so far may be combined with one another resulting in a ***DDP-distributed delay*** (see /Ciardo 95/). This type of delay will be denoted by the same symbol as the discrete probability mass function with finite support (though it has infinite support). Due to the closure property of DDP distributions with respect to finite weighted summing (positive weights summing up to 1), finite convolution (sum of DDP distributed random variables) and infinite geometric summing (sum of independent identically DDP-distributed random variables with a geometrically distributed number of addends), all such

1. In the following, we will usually not distinguish between geometric delay and modified geometric delay.

2. Note that the constant delay is a special case of this type of probability distributions, but the geometric delay is not due to its infinite support.

compositions of DDP-distributed random variables are again DDP distributed random variables. As a particular consequence, the symbols ▮ and ▯ are also special cases of the symbol ⊟, and all rules for compression given above with the symbol for a constant delay are also valid with the thick bar replaced by ⊟ .

Timeless transitions governed by the "may"-firing rule will be represented by square boxes, i.e. by □ , as is generally usual in Petri nets without time.

Note: The thick bar symbol as well as the rectangular box with the same shape are also used in the stochastic Petri net classes GSPN and DSPN, but without reservation of tokens: i.e. the tokens to be delayed by the timed transitions "wait" on the places before a timed transition until the delay is over - and may even be "stolen" by transitions in conflict. In our model, timed transitions carry a token: i.e. all tokens are consumed in the moment of enabling from the preplaces and fired onto the postplaces after the delay. ♦

As an example of how this step by step abstraction works, consider once more the weakly reduced net model of the service12 process (see Figure 5).

Before considering this process *separately* we should remark that due to its non-free-choice structure, the subnet defined by the set of nodes {S12.receive.10, S12.receive.16, S2.receive.9, chan1, chan2, S12.begin.wait.8, S2.receive.9} (see Figure 4 and Figure 6, or Figure 13 - in the latter case the place names S12.begin.wait.8 and S2.receive.9 have to be replaced by s12 and s2) has to be considered as one basic object with all of its transitions governed by the "may" firing rule. Here it becomes obvious that for a quantitative evaluation of the modelled system a common (conditional on the marking of chan1, chan2, S12.begin.wait.8 and S2.receive.9) probability distribution for the firing of the transitions of this basic object is necessary. Based on this distribution and on the parameters for the involved DDP-distributions (both have to be estimated based on observations and/or code analysis), the cycle times of the service processes and the overall conditional distribution for the service process can be simulated or computed, the latter thanks to the analytical possibilities of the DDP distributions. In particular it comes out that both service processes should be considered in combination, i.e. in one net object with a common time axis. (There, of course, the necessary delay and probability parameters of the two service processes should be considered independently.) For the sake of conciseness, we will consider only the service12 process in more detail here.

As follows from the considerations given above, the delay times from firing of the transition S12.receive.10 or of the transition S12.receive.16 until the token reaches the place S12.while.6 again are DDP-distributed. The particular form of these distributions can be obtained by a systematical abstraction from the weakly reduced net model. According to general experiences and measurements, only the "do" statements require significant time (modelled on the lowest level by constant delay transitions) whereas the time consumption of all other statements ("begin", "end", "for", "if"..., and any communication statements) can be neglected and therefore be modelled by immediate transitions. The branching probabilities for leaving the "for.11"-loop and for passing the "else.19" branch are assumed to be known too. Then we get a net of the form given in

Figure 11. A second (trivial) reduction step using the reduction rules given above would yield the net shown in Figure 12.

When the same reduction methodology is also applied to the requests and to the service2 processes, the overall view of the process system will be as shown in Figure 8.

6.3 General Proceeding

An appropriate application of MONs for quantitative analysis of parallel systems, which is intended as part of the approach presented here, consists of the following steps.

First, decompose the nets generated from the software into free-choice and non-free-choice objects as described above. Second, obtain the necessary constant delay s and probabilities by (generally local) testing and monitoring parts of running prototypes or by estimation. Third, carry out a (preferably parallel) simulation of the complete MON taking into account the obtained quantitative information. Alternatively, the use of appropriate numerical iterative methods might be useful.

The desired quantitative parameters are then predicted or assessed on the basis of the results obtained from the simulation or computation. Advantages of this approach as compared with the traditional GSPN or DSPN approach are an expected increased simulation efficiency and doing without a global time axis (in the general case). Additionally, in the MON models the "problematic" parts of programs and specifications (characterized by the corresponding non-free-choice objects, especially those causing confusion) become obvious as a result of the systematic aggregation of the free choice subnets into delay objects.

In order to guarantee a broad range of methods, thus permitting comparisons of their efficiency and accuracy, our approach intends to include the use of "conventional" tools such as GreatSPN /Chiola 91/ and TimeNET /German 94/ in addition to the "generic" MON-simulation.

7 Final Remarks

Petri nets provide an adequate common basis for a general workstation that extensively supports different methods of dependable parallel software engineering. Petri nets are a suitable intermediate representation for

- different languages,
- different phases of the software development cycle, and
- different validation methods.

Our investigations are greatly influenced by the goal of providing a workbench which can be applied by an engineer engaged more in software quality assurance than in Petri net theory. However, a thorough software validation is expensive and requires an adequate mathematical foundation.

In our opinion, the level of practicability of a method like this is strongly influenced by the level of engineer-like preparation of the tool kit provided. To fill the gap between theory and application, a lot of work is still to be done to support engineers in handling the underlying model in an effective and easy way. An ongoing prototype implementation of a graphic-oriented tool kit supporting the proposed validation methodology is pursuing these objectives.

The Petri net based software validation as an analytical approach suffers from the disadvantage that it is always done a posteriori. Independent of the validation success, the following advantages of the approach are seen.

- Any validation method directs the programmer towards a rethink of the program design.
- The lessons learned trying the validation can be turned into useful hints on how to construct parallel programs in general, and help us to better understand the problems inherent in parallel programs.

We claim these to be useful steps towards extending the so-called theory of structured programming (of sequential systems) to a theory of structured programming of (dependable) parallel systems, providing a discipline to design a priori distributed programs with certain useful properties. But any design discipline can guarantee proper properties only if it is applied properly. So finally, a thorough validation process including excessive testing is still necessary.

The application of the method presented here to a realistically sized example /Lewerentz 95/ is in preparation. This case study aims at Petri-net based development and step by step validation of the control software of a production cell. The cell consists of seven loosely coupled machine controllers acting independently of each other to a high degree. The qualitative validation is outlined in /Heiner 95/. Based on these results, we are now going to evaluate time-dependent properties.

Acknowledgement

We wish to thank Nick Grindell for correction and improvement of the English text.

8 References

/Balbo 92/

Balbo, G.: Performance Issues in Parallel Programming; in: K. Jensen (Ed.): Application and Theory of Petri Nets 1992, LNCS 616, Berlin: Springer, 1992, pp. 1-23.

/Chang 89/

Chang, C. K. et al.: Integral: Petri Net Approach to Distributed Software Development; Information and Software Technology 31(89)10, pp. 535-545.

/Chiola 91/

Chiola, G.: GreatSPN 1.5 Software Architecture; in: Proc. 5th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation, Torino, Italy, 2/1991, pp. 117-132.

/Ciardo 95/

Ciardo, G.: Discrete-time Markovian Stochastic Petri Nets; in: Stewart, W. J. (Ed.) Numerical Solution of Markov Chains ‘95, Raleigh, NC, January 1995.

/Clarke 86/

Clarke, E. M.; Emerson, E. A.; Sistla, A. P.: Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications; ACM Trans. on Programming Languages and Systems 8(86)2, pp. 244-263.

/Czichy 92/

Czichy, G.: Implementation of a Petri Net Generator for INMOS C Programs (in German); Techn. Report of Practical Studies, GMD/FIRST, Berlin 2/1992.

/Czichy 93/

Czichy, G.: Design and Implementation of a Graphical Editor for Hierarchical Petri Net Models (in German); Diploma Thesis, TU Dresden und GMD/FIRST, 6/1993.

/Gait 86/

Gait, J.: A Probe Effect in Concurrent Programs; Software Practice and Experience 16(86)3, pp. 225-233.

/German 94/

German, R.; Kelling C.; Zimmermann, A.; Hommel, G.: TimeNET - A Toolkit for Evaluating Non-Markovian Stochastic Petri Nets; Techn. Universität Berlin, Dept. of Informatics, Report 1994-19.

/Grzegorek 91/

Grzegorek, M.: Further Development of a PDL/D Compiler (in German); Techn. Report of Practical Studies, IIR/AdW, Berlin 1/1991.

/Halme 95/

Halme, J. et al.: PROD Reference Manual; Helsinki Univ. of Technology, Digital Systems Laboratory, June 1995.

/Heiner 88/

Heiner, M.: A Complexity Measure of Distributed Programs; Proc. 2nd Int. Seminar on Modelling and Performance Evaluation, Wendisch-Rietz, 11/1988, Informatik-Reporte/IIR 17/88, pp. 72-83.

/Heiner 92/

Heiner, M.: Petri Net Based Software Validation, Prospects and Limitations; ICSI-TR-92-022, Berkeley/CA, 3/1992.

/Heiner 94/

Heiner, M., Ventre G., Wikarski, D.: A Petri Net Based Methodology to Integrate Qualitative and Quantitative Analysis", Information and Software Technology 36 (7) 1994, pp. 435-41.

/Heiner 95/

Heiner, M.; Deussen, P.: Petri Net Based Qualitative Analysis - A Case Study; to appear as Techn. Report BTU Cottbus, 1995.

/König 85/

Koenig, H.; Heiner, M.; Onisseit, J.: Defining Report of the Protocol Description Language PDL (in German); Techn. Univ. of Dresden, Dept. of Informatics, Research Report TU 08 RS-L/LN-K5/015, 1985.

/Lewerentz 95/

Lewerentz, C.; Lindner, T.: Formal Development of Reactive Systems - Case Study Production Cell; LNCS 891, 1995.

/Lindemann 94/

Lindemann, C.: DSPNexpress: A Software Package for the Efficient Solution of Deterministic and Stochastic Petri Nets; Performance Evaluation, 1994.

/Marsan 87/

Ajmone Marsan, M.; Chiola, G.: On Petri Nets with Deterministic and Exponentially Distributed Firing Times; in: G. Rozenberg (Ed.) Advances in Petri Nets 1986, LNCS 266, Berlin: Springer,1987, pp. 132-145.

/Starke 90/

Starke, P. H.: Analysis of Petri Net Models (in German); Stuttgart: B. G. Teubner, 1990.

/Starke 92/

Starke, P. H.: INA - Integrated Net Analyser, Manual (in German); Berlin 1992.

/Whitworth 93/

Whitworth, J.: The Structure of Object Nets; Preprint, School of Computing, Staffordshire University, Stafford, UK, February 1993.

/Wikarski 90/

Wikarski, D.: Object Nets - A Canonical Class of Models for Behaviour Simulation and Structure Synthesis of Distributed Systems; in Proc. 3rd Int. Seminar on Modelling, Evaluation and Optimization of Dependable Computer Systems, Wendisch Rietz, 11/1990, Informatik-Informationen-Reporte 6(90)12, pp. 91-100.

/Wikarski 92/

Wikarski, D.: Locally Markovian Object Nets - The Conceptual Model; in Burkhardt, H.-D.; Starke, P.H.; Czaja, L. (Eds.) Proc. Int. Seminar on Concurrency, Programming and Specification (CSP 92), Humboldt-University Berlin, Berlin 11/1992, pp. 182-189.

/Wikarski 94/

Wikarski, D.: Modular Nets and Object Nets: A Unifying View; in Czaja, L.; Burkhardt, H.-D.; Starke, P.H. (Eds.) Proc. Int. Workshop on Concurrency, Programming and Specification (CSP 94), Humboldt-University Berlin, Berlin 36/1994.

9 Appendix (Figures)

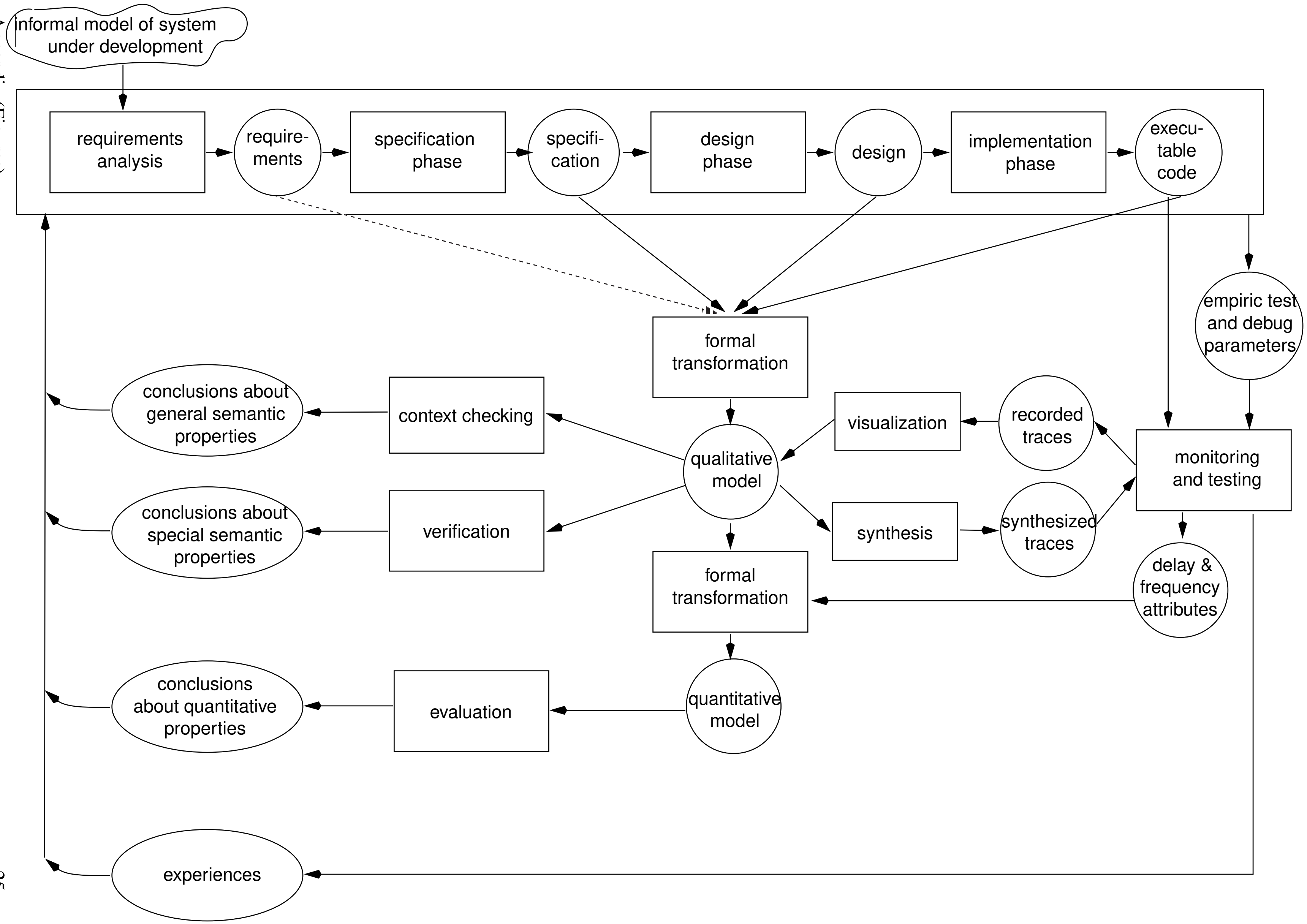


Figure 1: Overview of the net-based methodology.

Figure 2: Basic Petri net components used by the Petri net Generator.

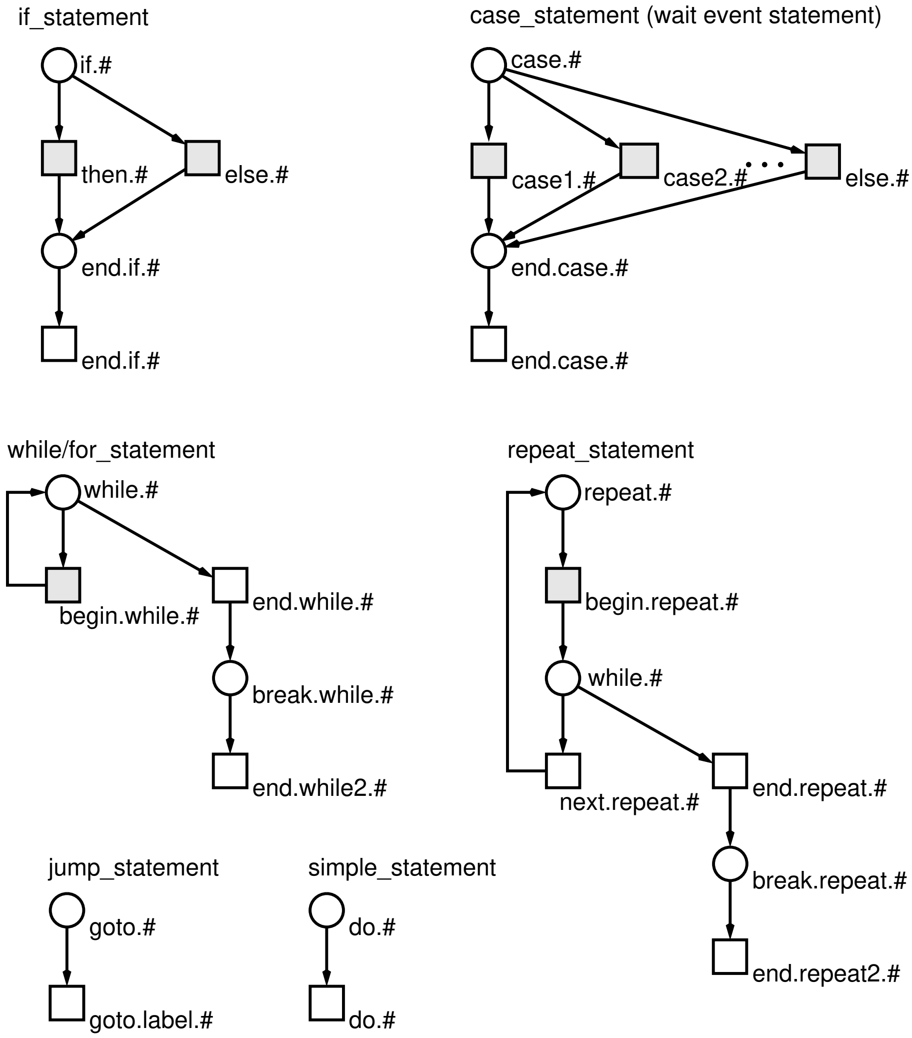


Figure 3: Source text skeleton of the running example.

```
main()          /* requests process */
{
    message chan1, chan2;               /* communication objects */
    int k, l;

    while(1)
    {
        produce_k();
        send k to chan1;                /* chan1!k */
        produce_l();
        send l to chan2;                /* chan2!l */
    }
}
```

```
main()          /* service12 process */
{
    message chan1, chan2;               /* communication objects */
    int x, i;

    while (1)
    {
        wait event                      /* alternative waiting */
        {
        receive x from chan1:           /* chan1?x */
            for (i=x; i>0; --i)
            {
                for_action();
            }
            break;
        receive x from chan2:           /* chan2?x */
            if (x)
                then_action();
            else
            {
                else_action1();
                else_action2();
            }
        }
    }
}
```

```
main()          /* service2 process */
{
    message chan2;                      /* communication object */
    int x, y, i;

    while(1)
    {
        calculate_x();
        receive y from chan2;           /* chan2?y */
        evaluate_y();
        for (i=1; i<x; ++i)
        {
            if (y)
                then_action();
            else
            {
                else_action1();
                else_action2();
            }
        }
    }
}
```

Figure 4: Fine-grained control structure model of the service12 process.

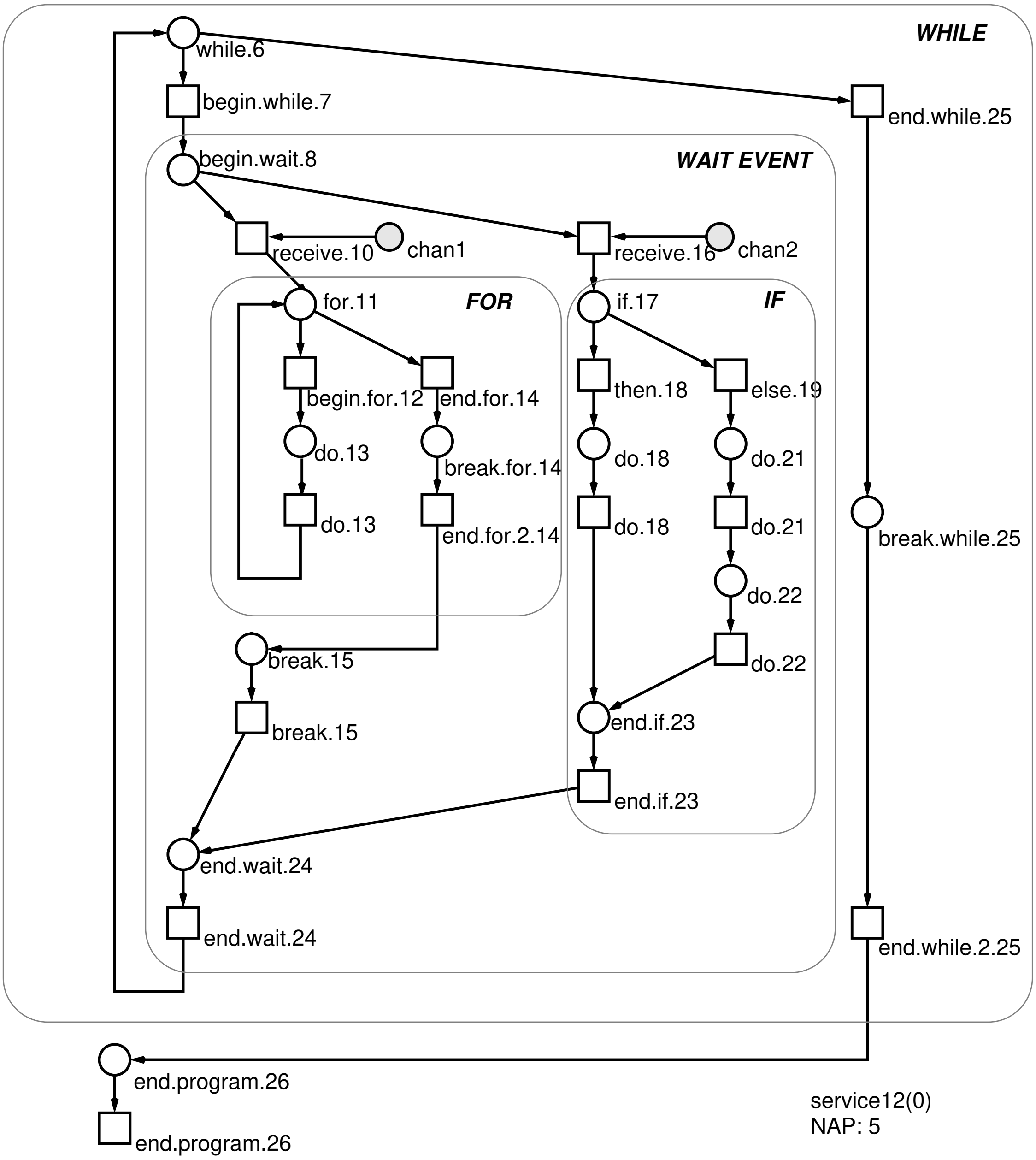


Figure 5: Statement skeleton of the service12 process.

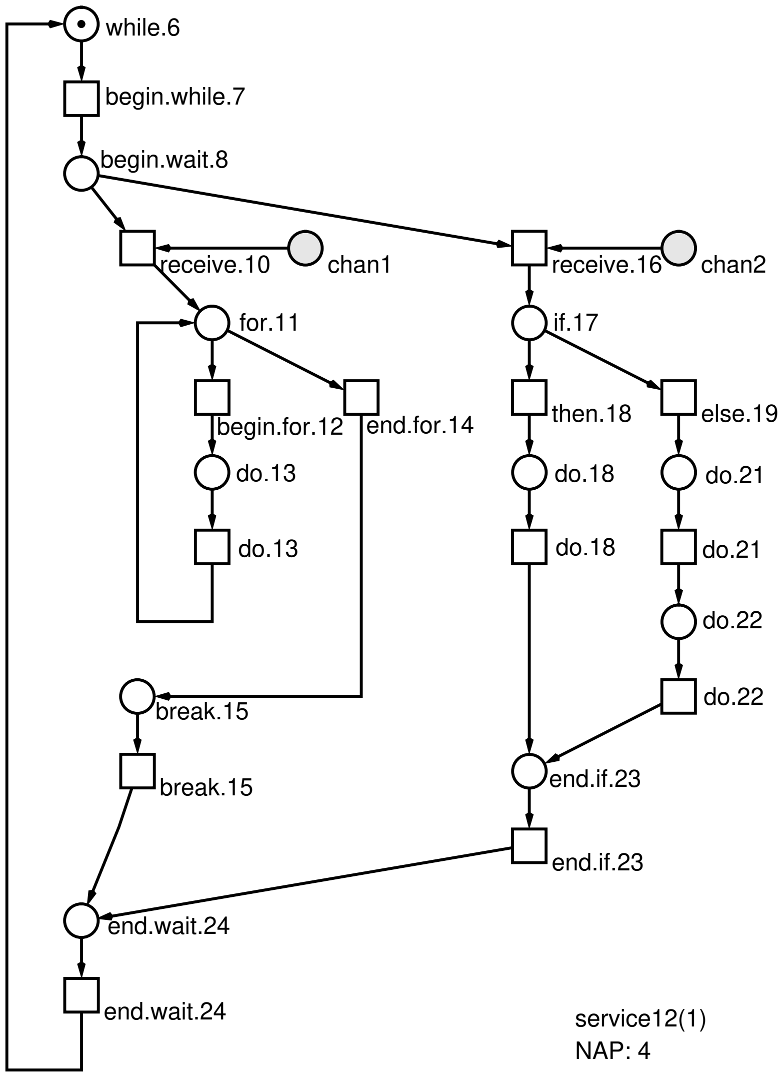


Figure 6: a) Statement skeleton of the service2 process.

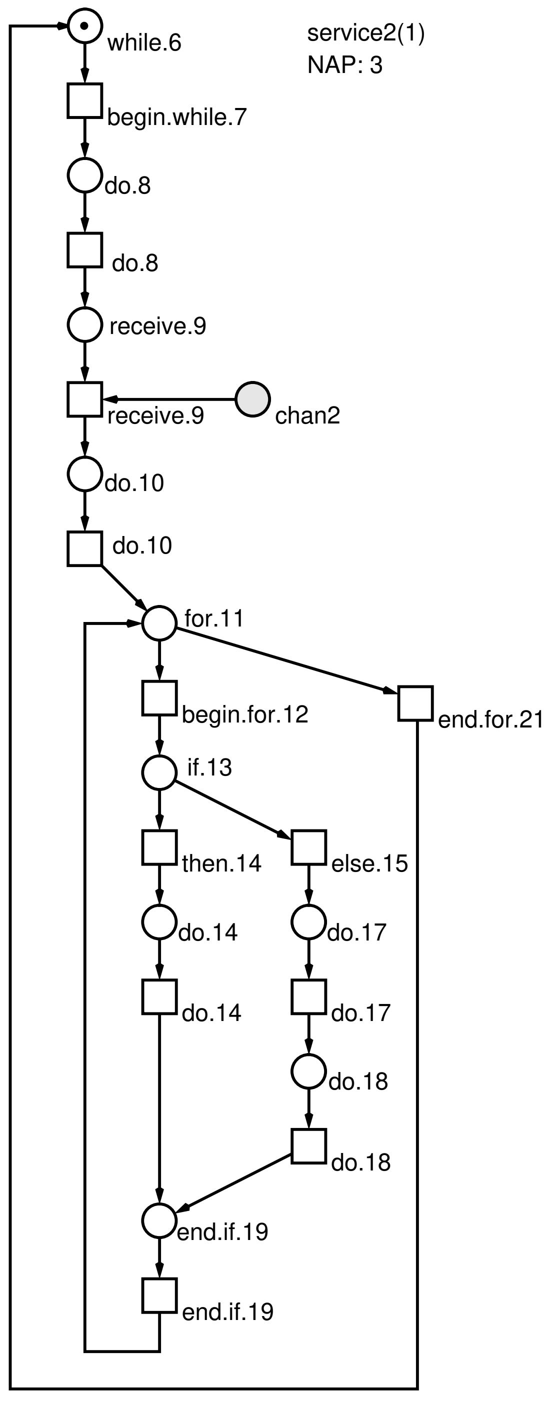


Figure 6: b) Statement skeleton of the requests process.

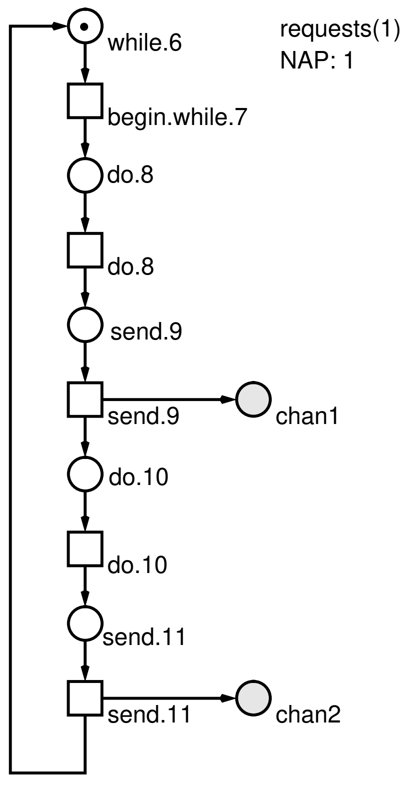


Figure 7: Top level of the hierarchically structured statement skeleton of the system.

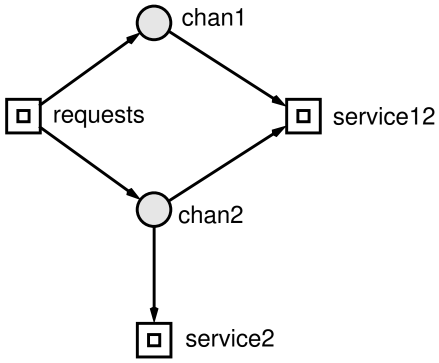


Figure 8: Communication skeleton of the process system.

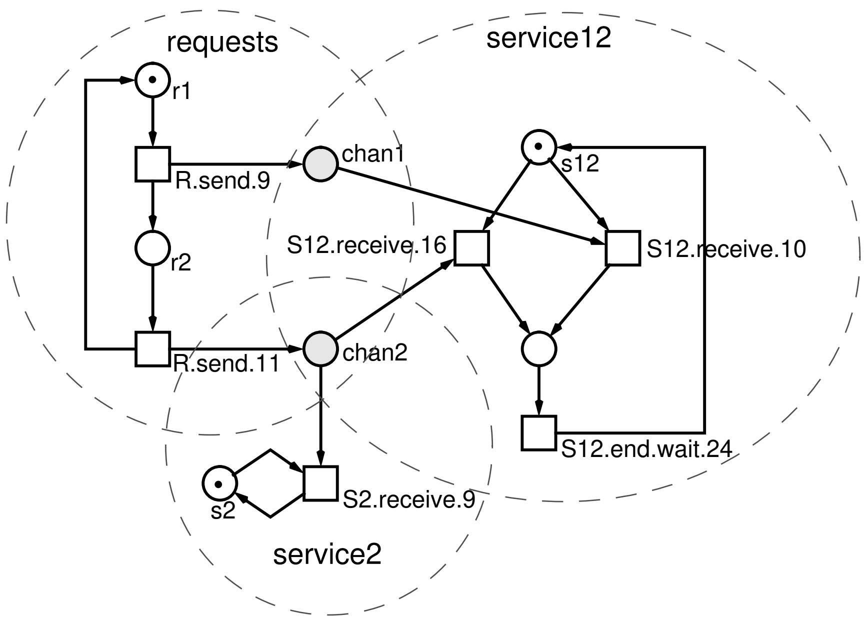


Figure 9: Strongly reduced system net.

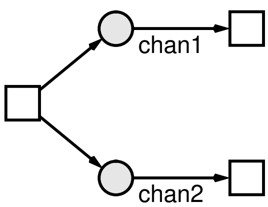
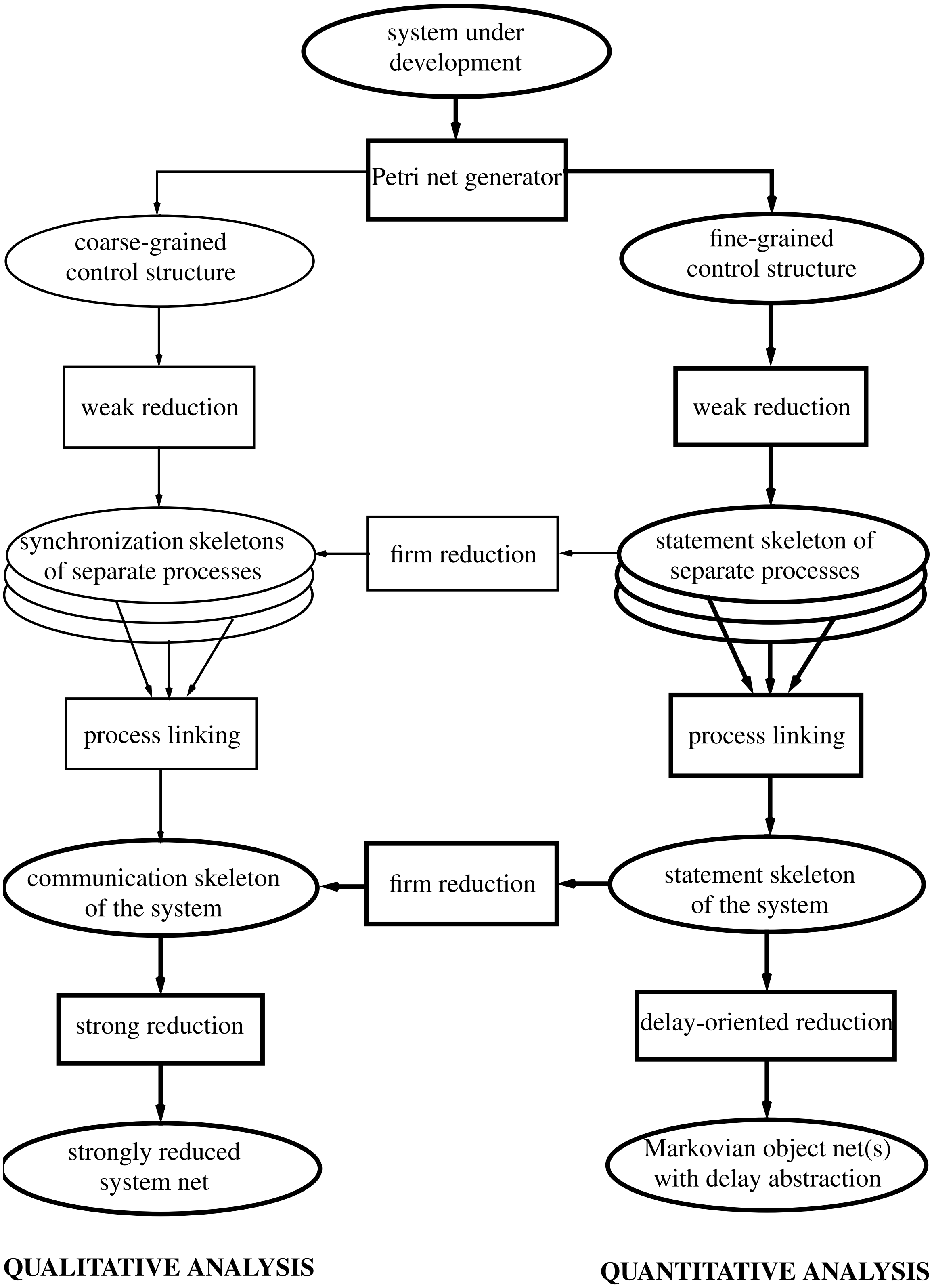


Figure 10: Reduction steps and its interrelations.



Remark: The steps and states arising in the example are represented with bold lines.

Figure 11: The service12 process after a first delay abstraction step.

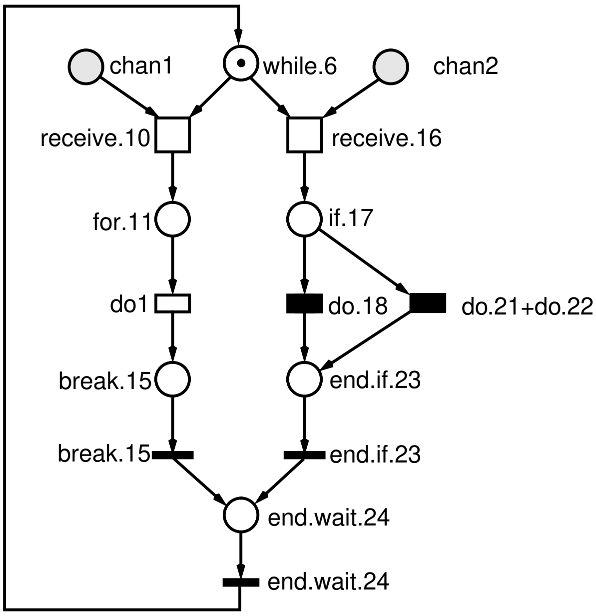


Figure 12: The service12 process after a second delay abstraction step.

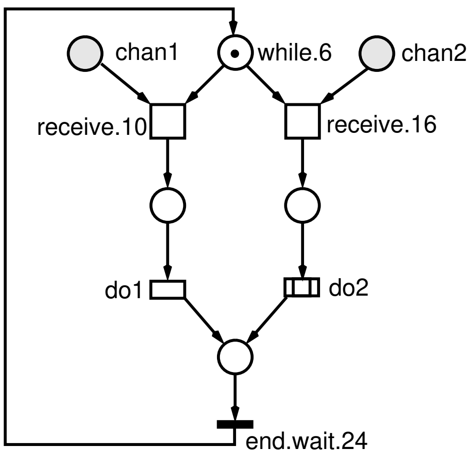


Figure 13: The overall net structure for combined quantitative and qualitative evaluation