Petri Nets for Reverse Engineering

DISSERTATION

DER WIRTSCHAFTSWISSENSCHAFTLICHEN
FAKULTÄT
DER UNIVERSITÄT ZÜRICH

zur Erlangung der Würde
eines Doktors der Informatik

vorgelegt von

WALTER KELLER

von

Herisau AR

genehmigt auf Antrag von

PROF. DR. LUTZ H. RICHTER

PROF. DR. HELMUT SCHAUER

OKTOBER 2000

Die Wirtschaftswissenschaftliche Fakultät, Abteilung Informatik, gestattet hierdurch die Drucklegung der vorliegenden Dissertation, ohne damit zu den darin ausgesprochenen Anschauungen Stellung zu nehmen.

Zürich, den 28. Juni 2000 *

Der Abteilungsvorsteher: Prof. Dr. P. Stucki

* Datum der Promotionsfeier

Abstract

The aim of this work is to conduct research into synergies between Petri-net theory and reverse engineering. The existence of such synergies is not obvious because each sector is based on different assumptions. These differences relate to two modelling paradigms: clustering and folding.

Clustering merges neighboured nodes and corresponds to the construction of complex systems from subsystems. It is widely used in software engineering and in practical applications of Petri nets. Foldings only merge transitions with transitions, places with places and arcs with arcs. They group similar functionality. Hence they preserve behaviour, allow the transfer of semantics and provide deep theoretical insights by means of far-reaching connections to other models of concurrency.

A folding-based Petri-net algorithm for reverse engineering is introduced. It recovers a coloured net from an unstructured flat Petri net. The two nets are connected by a folding which amounts to a compact specification of the source net. The algorithm is both flexible and scalable, and this work shows how application heuristics can be integrated into it. Its cost is almost linear with respect to the size of the input net, which is remarkable in the field of reverse engineering.

Petri nets may serve as an intuitive model of the interplay of the structural, functional and dynamic aspects of a system. Various methods of modelling aspects apart from concurrency by Petri nets represent an innovation. With such a translation, the algorithm may also analyse legacy systems outside the realms of Petri nets. The result is a novel and powerful method of reverse engineering. A specific example shows how a high-level design may be recovered from low-level implementation information. Moreover, the recovered colouring contains a complete specification inclusive of the data model.

The reverse engineering part of this work concentrates on folding-based Petri-net methods because they contain new features. On the other hand, clustering-based techniques share many similarities with known methods. For best results, however, clustering and folding should be appropriately combined. The foundations for such combinations are laid down in the first part of this thesis.

Many Petri-net classes known from the literature may be grouped into folding-based and clustering-based types. However, no well-defined link exists between them which would allow the strengths of both approaches to be combined, especially for practical applications.

Such a link is presented here in the form of an adjunction, which is a strong two-way relationship taken from category theory. It links folding-based and clustering-based categories. It is shown that these categories have properties typical of folding-based and clustering-based Petri-nets respectively. Further compatible adjunctions express the Petri-net dichotomy of structure and behaviour. To the best of the author's knowledge, this basic principle of Petri-net theory has not yet been formulated categorically.

For practical applications, it is important that these concepts can be integrated fairly easily with existing Petri-net tools. This will enrich them with the power of a categorical machinery, e.g. with morphisms, universal constructions and the transfer of behaviour. Coloured nets are simply defined as special comma categories, i.e. essentially folding morphisms. The reduction algorithm introduced here is a proof of the practical value of this approach: it is an iteration of couniversal constructions and the reduction itself has couniversal properties.

Keywords

Petri nets; reverse software engineering; clustering; folding; structure; behaviour; semantics; occurrence systems; category theory; design metaphor.

Zusammenfassung

Das Ziel dieser Arbeit ist die Erforschung möglicher Synergien zwischen Petri-Netz-Theorie und Reverse Engineering. Dass solche Synergien möglich sind, liegt nicht auf der Hand, zu verschieden sind die Grundannahmen in den beiden Gebieten. Diese Gegensätzlichkeit bringe ich mit zwei unterschiedlichen Modellierungs-Paradigmen in Verbindung, nämlich mit Folding und Clustering.

Clustering gruppiert Nachbarschaften und entspricht der ingenieurmässigen Konstruktion von komplexen System aus Teilsystemen. Es ist somit ein Grundprinzip von Reverse Engineering und unabdingbar für jedes Praxis-taugliche Petri-Netz-Werkzeug. In Petri-Netzen kombinieren Foldings Stellen mit Stellen, Transitionen mit Transitionen und Kanten mit Kanten. Sie gruppieren somit ähnliche Funktionen, ermöglichen es Verhaltenseigenschaften zu übertragen und haben weitreichende Verbindungen zu anderen Modellen der Nebenläufigkeit.

Diese Arbeit führt einen folding-basierten Petri-Netz Algorithmus für Reverse Engineering ein. Kernstück ist die Reduktion eines unstrukturierten Netzes in ein gefärbtes Netz. Die beiden Netze sind durch einen Folding-Morphismus verbunden, der eine kompakte Beschreibung des Ursprungsnetzes beinhaltet. Der Grundalgorithmus ist anpassungsfähig, z.B. kann er mit verschiedenen Anwendungsheuristiken kombiniert werden. Hervorragend für einen Reverse-Engineering-Algorithmus ist die Effizienz. Die Kosten sind beinahe linear in der Grösse des Ausgangsnetzes.

Petri-Netz-Modelle können ein kompaktes und intuitives Bild des Zusammenspieles von strukturellen, funktionalen und dynamischen Aspekten eines Systems zeigen. Neu daran ist es, Petri-Netze für Fragen, die wenig mit Nebenläufigkeit zu tun haben, zu verwenden. Mit geeigneten solchen Übersetzungen wird der Reduktionsalgorithmus allgemein anwendbar, nicht nur für Petri-Netze. Das ergibt eine neue, mächtige Reverse Engineering Methode. An einem konkreten Beispiel wird gezeigt, wie aus elementaren Implementations-Informationen eine abstrakte Architektur wiedergewonnen werden kann. Die berechnete Färbung des reduzierten Netzes spezifiziert den Zusammenhang von Architektur und Implementation und spiegelt das Datenmodell wider.

Der Reverse-Engineering-Teil dieser Dissertation befasst sich mit folding-basierten Methoden, weil sie neue Möglichkeiten enthalten, während

clustering-basierte Petri-Netz-Methoden viele Ähnlichkeiten mit bekannten Reverse-Engineering Techniken teilen. Aber selbstverständlich soll nicht Folding gegen Clustering ausgespielt werden, sondern ein sinnvolles Zusammenspiel gesucht werden. Theoretischen Grundlagen dafür werden im ersten Teil dieser Dissertation gelegt.

Viele aus der Literatur bekannte Klassen von Petri-Netzen können in folding- und clustering-basierte unterteilt werden. Was jedoch fehlt, ist eine wohldefinierte Verbindung zwischen den beiden Gruppen. Erst eine solche erlaubt die Stärken beider Ansätze zu kombinieren - auch in praktischen Anwendungen.

Eine solche Verbindung wird hier vorgestellt und zwar in der Form einer Adjunktion - einer starken Zweiweg-Beziehung aus der Kategorientheorie. Es wird gezeigt, dass die verbundenen Kategorien die typischen Stärken von clustering- bzw. folding-basierten Petri-Netz-Klassen aufweisen. Darauf aufgebaute Kategorien und Adjunktionen bilden eine Formalisierung der Dichotomie von Struktur und Verhalten - einem grundlegenden Petri-Netz Prinzip, das meines Wissens noch nie kategorientheoretisch formuliert worden ist.

Für die Praxis von Bedeutung ist, dass diese Konzepte relativ einfach in bestehende Petri-Netz-Werkzeuge integriert werden können. Das ermöglicht diese durch kategorische Mittel wie Morphismen, universelle Konstruktionen oder Übertragung von Semantik zu bereichern. Gefärbte Netze werden verblüffend einfach definiert, nämlich als spezielle Kommakategorien, d.h. im wesentlichen als folding-basierte Morphismen. Ein Beispiel für den praktischen Wert dieses Ansatzes ist der eingeführte Reduktionsalgorithmus: Er ist eine Iteration von couniversellen Konstruktionen und die berechnete Reduktion hat selbst wieder couniverselle Eigenschaften.

Contents

Document info. Version 2.4, 03.12.00 15:22, 294701 characters, 59698 words, 157 pages.

1 Introduction

The effort to understand existing software systems is known to be a major contributor to software costs. Experience has shown that every new programming technique brings new maintenance problems and needs specific reverse-engineering methods. The use of Petri nets has emerged in recent years and the need to reverse-engineer such systems will inevitably arise. This implies the importance of research into methods of reverse-engineering Petri nets. However, reverse engineering is intrinsically difficult. There are still many situations in which traditional analyses fail and in many domains reverse engineering remains an art rather than an engineering discipline. This justifies the exploration of radically different methods. This work represents a possible response to these two challenges:

Synergies between the fields of reverse engineering and Petri nets

Research on this topic yielded two conflicting structuring principles: folding and clustering.

In clustering, directly connected entities are grouped together. Neighbourhoods are collapsed into a single object which also swallows relationships. Foldings, on the other hand, merge similar objects and, separately, similar relationships. They group functionality and not adjacency. A model object stands for similar objects and hence inherits many properties.

Clustering respects vicinity, is the usual method of breaking complex systems down into subsystems and is consequently predominant in reverse engineering. But a clustered node is neither a transition nor a place and is incompatible with the rigorous semantics of Petri nets. Folding is therefore preferred for theoretical purposes but clustering is indispensable in practice. Furthermore, coloured Petri nets are a very convenient representation of foldings and are important for practical applications.

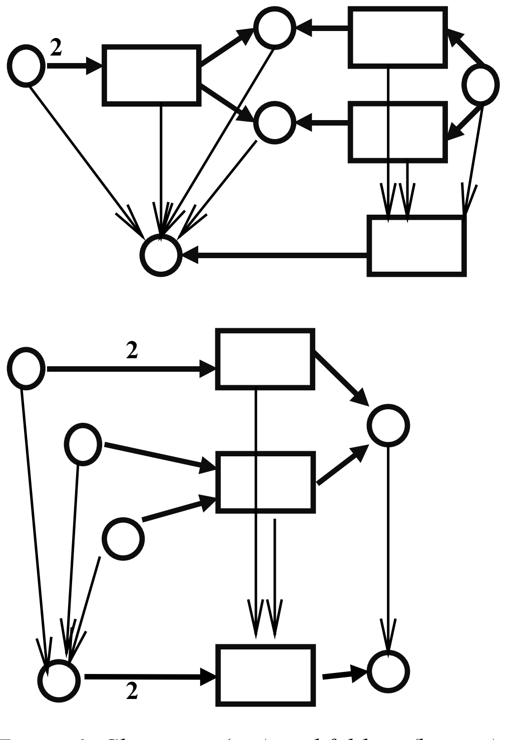


Figure 1. Clustering (top) and folding (bottom)

The research conducted for this thesis highlighted the importance of this concept:

The dichotomy of folding and clustering

In terms of this dichotomy, the main results may be stated as:

- A folding-based Petri-net algorithm for reverse engineering is introduced. It recovers a coloured net from an unstructured flat Petri net; it is also flexible and scalable.
- Several ways of modelling structural, functional and dynamic aspects of a system by Petri nets are elaborated. They allow this algorithm to be applied as a novel and powerful reverse-engineering method outside the realms of Petri nets. It recovers high-level design and specification information such as the data model from a legacy system.

- Petri-net categories which form a bridge from folding to clustering are defined. Within these categories, this algorithm represents an iteration of universal constructions.

The algorithm inherits many properties from universal constructions such as uniqueness and the possibility of performing optimisations. The almost optimal performance is especially remarkable as the field of reverse engineering is notorious for huge confusing search spaces and combinatorial explosions.

The second point shows that Petri nets may be used as an intuitive and concise model of a system and we therefore propose to use them as design diagrams. Together with the first point, we conclude that folding-based Petri-net methods qualify as a powerful method for both forward and reverse engineering. However, best results would be expected from an appropriate co-operation of clustering and folding.

The third point lays a theoretical foundation for such a co-operation. Existing Petri-net classes are either clustering-based or folding-based, but connections between the two paradigms are missing. The bridge from clustering to folding is actually an adjunction, which is a strong relationship taken from category theory. The categories introduced support graph-based and linear algebraic methods and allow coloured nets to be defined in a way which is simple but nevertheless encompasses the net-theoretical essence of many classes of coloured Petri nets containing these attractive diagrams with their node colours and arc inscriptions. They allow practical applications to benefit more easily from the deep theoretical insights offered by folding-based Petri-net categories and from the power of categorical machinery.

1.1 Petri Nets

For our purposes, existing Petri-net formalisms may be classified in two groups:
- Clustering-based - these usually make use of vicinity-preserving graph morphisms. System compositions using clustering and graph-based methods are supported.
- Folding-based - these consider a Petri net as a two-sort algebra and pre and post as unitary operators which must be respected by morphisms. Behaviour is transferred smoothly and powerful links are set up to many other models of concurrency, thus yielding deep theoretical insights.

We will introduce a typical category for each of these two groups. Place-transition nets (PTNET) are clustering-based. Their form of clustering is very useful for software engineering and their morphisms are compatible with linear algebraic Petri-net techniques. FNET, the category of nets with foldings, is a subcategory of PTNET.

The bridge from clustering to folding is formed by two compositional adjunctions which connect the two categories over PPNET, the category of nets with place-preserving morphisms. Notice that an adjunction is a two-way relationship, and is much stronger than a simple functor. On a theoretical level, these adjunctions clarify the relationships between the two groups of Petri-net classes specified above. In practice, they allow clustering nets to be simulated by folding nets with reasonable computational effort.

Simple categorical methods extend nets to ***coloured and hierarchical nets***. They are connected by a web of adjunctions which is compatible with the dichotomy of clustering and folding. Although highly abstract, these categories are tightly related to existing classes of coloured nets. Existing net tools could thus benefit by virtue of our new categories from the power of categorical machinery, e.g. morphisms for simulation or implementation/design relationships, universal constructions for flexible compositions of subsystems, and functors for the transfer of semantics from one environment to another.

In both PPNET and FNET, ***all universals and couniversal constructions exist***. This allows nets to be composed from subnets in a flexible and natural way, and graph transformation techniques etc. to be applied. Furthermore, we will show that an iteration of couniversal constructions may be used for reverse engineering, namely in the form of the recovery of a coloured net from an unstructured net. The computed reduction has many of the powerful properties typical of universal constructions, including uniqueness. Also, the clear theoretical formulation permits flexible variations and important optimisations. This is especially remarkable in the field of reverse engineering, which is notorious for huge confusing search spaces and combinatorial explosions.

A Petri net is a static structure, merely representing a bipartite graph. If an initial marking is added, a token game may be played. This is an important principle of Petri-net theory:

The dichotomy of structure and behaviour

However, this dichotomy has not yet, to the author's knowledge, been formulated in terms of category theory. Even worse, morphisms in conventional folding-based Petri net categories lose their relationship to the underlying graph. This work succeeds in defining behavioural categories (PTSYS, PPSYS and FSYS) which are connected by coreflections (special adjunctions) to net categories. Moreover, this procedure yields a web of seven adjunctions categorically connecting the dichotomy of structure and behaviour to the dichotomy of clustering and folding.

Strong relationships to semantics are typical for folding-based Petri net classes. This is also true here, thus confirming that FNET is folding-based in a deep sense. The unfolding of a system to the semantics of its step reachability is functorial. It consequently shows how morphisms transfer behavioural properties such as liveness. Process-based semantics expressed in terms of occurrence nets turns out to be a coreflection to a subcategory of FSYS which is similar to previous results. In order to achieve a coreflection to the whole category of FSYS, a generalisation of weighted occurrence systems is introduced. It is proposed as an alternative to safe occurrence systems which yields a purer image of causality and branching.

1.2 Reverse Engineering

Given the author's professional background as a consultant for the more technical aspects of the development of mainframe-based commercial software in large IT departments, this work focuses on the reengineering of conventional applications. The aspects of concurrency are thus delegated in a standard way to the database management system and the transaction monitor. This means that ***Petri nets are used in a field where they cannot exploit their natural strength to model concurrency***.

As a consequence, a given system must first be translated to a Petri net. The net is subsequently analysed and the result translated back. However, it turned out that this is not a magician's trick to simplify the intricacies of reverse engineering. Rather, several such translation methods exist which yield a graphically attractive, concise and intuitive model of the structural, functional and dynamic aspects of a system. The reasons for the power of such modelling paradigms are, for instance:

- the separation of active and passive elements in transitions and places
- foldings in terms of coloured nets

An unforeseen result of this work is that we propagate ***Petri nets as a modelling metaphor*** for both forward and reverse engineering. This thesis does not exploit the Petri-net dialectic of structure and behaviour for reverse engineering, although a similar dialectic of static program source and dynamic program execution lies at the heart of software engineering. The specific algorithms operate only on the bipartite graphs. However, the metaphor of Petri-net behaviour was the driving intuition behind our research.

This work essentially concentrates on a particular reverse-engineering task: the recovery of a coloured net from an unstructured one. This is a ***folding-based Petri-net method***. A complete reverse-engineering framework must also support clustering-based analyses, the selection and combination of different reductions to a larger picture, etc.. We observed that folding-based methods offer many new features, whereas many traditional methods apply to the remaining tasks.

We are equipped for this task as regards the net-theory part with net categories and iterated universal constructions which build a unique reduction of a net. Nevertheless, we decided to implement a prototype of the algorithm in Smalltalk and to present it in this work together with the source code of its key parts. A major benefit of this approach was the opportunity to experiment with the algorithm on specific nets, which led to the discovery of surprising effects.

The basic idea may be illustrated by another use of the word folding. Just like folding a shirt, two adjacent parts which fit together are overlaid. The reduction works iteratively, i.e. it may overlay fitting parts whenever they become adjacent, and it stops when there are no more adjacent fitting parts. The algorithm offers three degrees of freedom:

- Similarity: How is similarity which allows two subnets to be merged defined?
- Adjacency: How is adjacency which allows two similar subnets to be merged defined?
- Choice: Which subset of the mergers allowed by the above two criteria is selected?

A first algorithm uses only the similarity criterion. It classifies the transitions of the net. Two transitions belong to the same class iff they form isomorphic subnets together with their surrounding places. Afterwards, larger subnets are classified. This yields a simple algorithm together with a feedback mechanism that decides which subnets are to be merged. It already yields a clear analysis. Unfortunately, there are two major problems: we could not find a way to improve the first result obtained, and it is a NP-hard problem to compose subnets and identify them in the source net.

Another approach uses an adjacency criterion instead of enlarging subnets. What looks like a simple algorithm with the usual recursive programming techniques turns out to require some sophisticated mechanisms - especially if high performance is needed. ***Performance is almost optimal*** – cost is nearly linear with respect to the size of the analysed net. This is a very attractive option in the field of reverse engineering which must fight against notorious combinatorial explosions. The performance results from a combination of locally merging subnets during graph traversal, the shift of the work from the origin net to the current reduction and several sophisticated data structures.

This version of the algorithm is flexible and allows ***integrating application heuristics***. We show this by the example of relationship cardinalities. They are applied twice: first the heuristics allows a more detailed reduction to be computed. Secondly, they enable us to colour the reduced net. Together, this procedure recovers both the original design diagram and a specification which relates design and implementation, including the data model. Indeed, the prototype implementation of the algorithm computed reverse-engineering diagrams which were better than those in the original design of the example application.

The algorithm analyses a net without pre-knowledge such as a cliché library of known programming constructs. This can be seen as an advantage as it is still an open question whether such a library may be constructed and used for real-world systems. However, an experiment indicates that the algorithm has the potential to detect reused components. It also offers a natural interface to such a cliché library where a need for it would arise.

These and further reflections show that the algorithm developed here represents a powerful reverse-engineering tool with many different applications. It ultimately leads to our thesis that folding-based Petri-net methods are a valuable tool for both forward and reverse engineering.

1.3 Document Structure and Conventions

To enhance readability and to clarify the major line of thought, the main topics are structured in this document as follows:

Continuous text	*Supplements*
3 Petri Nets	7 Appendix: Proofs and Details for Petri Nets
4. Reverse engineering	8. Appendix: Proofs and Details for Reverse Engineering

To facilitate navigation, corresponding chapters have the same decimal structure; e.g. the proofs for 3.4.2 may be found in appendix 6.4.2. Moreover, the reader may use forward and backward links. They are encoded by the following document-structuring symbols:

♦	marks the end of a proof, a remark, an algorithm etc.
⇨15	points to a reference on page number 15 . In Acrobat Reader, a mouse click on the page number triggers a jump to the reference. The page number is followed by a symbol that shows the type of reference.
⇨15◈	points from the main chapters to additional information in the appendix, e.g. from a proposition to its proof.
⇨15⇦	links back to where the reader probably originated. Thus it links back to the proved proposition at the end of a proof in the appendix.

2 Literature Review

Although abundant literature is available about Petri nets and reverse engineering, the intersection between the two fields is surprisingly small. The situation is complicated by the fact that we work in both subjects in a rather non-standard way. The consequence is that we cite many references but the relationship to the current work is sometimes rather weak.

The chapter starts with literature about Petri nets. First, Petri net models directly related to our work are presented. Some applications of nets for modelling of software systems follow. The next sections treat reverse engineering. We give references that show the state of the art and the practice as well as papers that compare to the present work. The last sub-chapter reviews the use of nets in reverse engineering.

2.1 Petri nets

For a general introduction and overview to Petri nets refer to [Rei98]. Additionally many of the cited books contain introductory chapters.

There are many definitions of net morphisms in the literature. Let's start with definitions maintaining strong relations to the underlying graph and often supporting clustering. [GLT79] defines morphisms as mappings of the nodes of nets that f: X .X' mapping neighbours in X to neighbours in X' and retaining the transition/place property, if the nodes are not collapsed. [Chr80] sketches the use of such morphisms for software engineering – this approach is extended in our work as a clustering technique. And in the outlook the article mentions the relationship between morphisms and invariants – a question answered in a more general setting in this work. [Des96] defines vicinity-respecting homomorphisms similarly and shows some basic structural properties for such morphisms. This definition is common for graphs and is used in this work as the graph component of a place-transition morphism.

A major inspiration for our work was [Feh93]. Here the same vicinity-respecting homomorphisms are used to define refinement and abstraction. A refinement is the source of a morphism that is surjective on the arcs and the nodes, whereas, the image is the abstraction. A graph of such refinements builds a hierarchical net and a collection of net morphisms forms a hierarchical morphism, making the hierarchical nets a category.

[Lak97] handles colours and morphisms in a similar way as we do. He uses much more complicated definitions for morphisms to get a narrow relationship between the two nets, but the paper does not work this out. A more crucial difference is, that the defining equations for morphisms are different:

- an occurrence of t is mapped to an occurrence of f(t) in our definition, but in [Lak97] in the other direction from t' to (a subset of) $f^{-1}(t')$.
- our definition gives distinct roles to transitions and places whereas [Lak97] tries to maintain a symmetry.

The other line of morphisms that we call folding-based considers a Petri net as a two sorted algebra – with multisets of places and multisets of transitions – pre and post as unary operators and the initial marking as a constant. A net morphism then is simply an algebra morphism that has to preserve the 3 operators. [Win84a], [Win84b] or [Win86] introduced the first such definition together with occurrence systems as the semantics, connecting the two categories by a coreflection. This needs some restrictions, most notably, only safe nets are considered.

The line of publications continuing this work e.g. [MMS92] used definition of a morphism f: N→N' like

- f is a map from T to T' and a linear multiset homomorphism from markings of N to markings of N'
- f pre = pre' f
- f post = post' f
- plus possibly some further restrictions

This definition disallows the collapsing of a node with its neighbours. Hence the usual clustering technique of software engineering is not supported. But the big advantage is that the relationship of a marked net with its semantics – e.g. reachability or occurrence nets – is functorial. This means a net morphism is translated compatibly into a morphism of the semantics. No such relationship is published for the clustering based morphisms. The unfolding of a coloured or hierarchical net into the flat is not a contradiction to this statement – it is rather an internal or definitional transformation rather than semantics.

In [MMS92] an adjunction between occurrence nets and Petri nets is constructed. We modified this construction to get two similar adjunctions between the categories developed in this work.

[Sas94] introduces a cube of 12 adjunctions between 8 categories modelling concurrency. The three axes of the cube correspond to the dialectics of system / behaviour, linear-time / branching-time and interleaving / non-interleaving. This cube does not contain a Petri net category but in appropriate categories the unfolding of a Petri net system to it's reachability graph gives an adjunction between transition systems and net systems as described in [NS98] and shown in [NRT90]. The connection to this beautiful cube really demonstrates that these purely folding based morphism definitions allow far reaching connections – making these categories very attractive from a theoretical point of view.

But the categorical framework also starts to attack more concrete tasks. For example [NC95] gives an elegant categorical description of many variations of bisimulation and other forms of behavioural equivalence. This opens the possibility to prove interesting properties of concrete nets within the language of category theory.

Folding based morphisms easily extend to algebraic high-level nets. [Pad98] uses categorical constructions in such nets to prove safety properties. This is used for real-world software development using a combination of Petri nets, algebraic specification and algebraic graph transformation.

2.2 Modelling Software by Petri Nets

Petri nets are often used to model workflow processes ([Aal97], [Obe96]), manufacturing plants ([Dic93], [Ezp98], [Zho98], and [Zim97]), embedded systems ([Ess97]), communication networks [APNC99] or performance issues ([Lin98]). In classical software engineering they are sometimes used to model the software engineering process ([Avr95]) or to model the dynamic aspects of a software system [Mai97]. But the functional aspects of software are seldom modelled by nets.

[Ell95] models the changing of workflows by Petri nets. Workflow promises easy changes. But what happens with tasks started in the old and finished in the new system? The paper wants it to be processed by a firing sequence that is valid either in the old or in the new system. Conditions to enforce this requirement are formulated.

[Mai97] proposes to use Object Coloured Petri Nets – an OO extension of the CPN formalism [Jen92] – instead of the usual state charts, collaboration diagrams etc.. These are promoted for example by the Unified Modelling Language UML [Rat97] to describe the dynamic aspects of systems. [Mai97] argues that these techniques are less expressive and lack a formal semantics of Object Coloured Petri Nets.

In [Fri97] data encapsulation and data abstraction are implemented in CPN ([Jen92]) using a single interface place. Graphically this interface place is painted as a big circle and the internals within this circle, which is an attractive visualisation of subsystem composition. The inside can be refined or instantiated in different ways, which gives a lot of flexibility.

[Mir94] translates behavioural specifications into Petri nets. Each operation in the specification is annotated with pre- and post-conditions. An operation is modelled by a place. A mark in the place signifies that the operation is in progress. The place gets an input transition, to start the operation. Control- and data-flow dependencies and the pre- and post-conditions are translated in predecessors of the start transition and successors of the place. To get a safe net, additional connection nodes might be necessary. Reductions on the resulting net optimise the implementation.

[Erm97] models medical patient data and hospital workflows by algebraic high-level nets. The interesting point for us is that push outs are used to construct large nets from simple ones. And it is shown that under certain circumstance properties from the small nets transfer to the combined net.

[Fel98] defines an implementation relation between timed Petri nets. The implementation relation is defined on the semantic level. Here the semantics of a net are the theorems of the theory obtained by translating a net to temporal formulas. Hence the procedure is to start from semantics and then look for net transformations (like refinements) that preserve the semantics. This is contrary to our approach which defines morphisms on a net level and then looks how they transfer semantics.

2.3 Reverse Engineering in Practice

Reverse Engineering is still a difficult business, and the usual case is, that only a combination of standard tools/methods with problem specific know how and ad hoc methods lead to some useful result. This fact is not clearly reflected in the literature. As in our paper, the successes for single cases are reported, the many trial and errors are hidden as research, bad examples or teething troubles of prototype software. One of the papers where this can be seen (more than read) directly is [Bur96]. The diagrams remain overloaded with edges till the end.

[Bel97] compares four reverse-engineering tools and [Sto97] observes users maintaining a program with three tools (overlap two). It seems that none of the tools could analyse deeper than to a call-, control- or data flow-graph. Not even impact analysis (transitive hull of control and/or data flow) was mentioned.

A typical reverse-engineering scenario is presented by the IDENT method [BUR98], [Bur97]. A call graph is generated from the source code, from this a dominance tree wherein candidate reusable units are identified and then brushed up in seven further steps.

Many of the most sophisticated reverse-engineering techniques are tailored for reengineering. The current top topics in the literature are fixes for year 2000 related problems and the wrapping of legacy code into objects.

[Luc97] proposes a six-step migration path, starting with static analysis, doing some architectural re-clustering, wrapping and finally migrating the wrapped objects to indigenous objects. This requires specialised analysis and guidance by the user. Additionally automatic sequences may be helpful. Other examples are [Sub96], which uses a different number of different steps, but also defines sequence.

The thesis [And96] describes a detailed procedure to transform real Cobol Applications (with Files, Database, Redefines etc.) into an OO-Model. Different graphs, relations, analysis techniques and heuristics are used to get first a relational scheme and then an ERC+ (Entity Relationship Model with Complex objects). Finally the OO schema is derived; uncertain conclusions are conciliated using constraint satisfaction techniques.

Some of these techniques like spreading sets have similarities with our approach to detect colours. The same holds for [Deu98]. She finds types in a Cobol source program. The method is to generate an equivalence relation over variables, generated by assignments, comparisons, arithmetic calculations and so on.

[Cim98] analyses legacy RPG Programs to wrap them into objects, that can be used in client server environments. The first phase identifies candidate objects by two criteria:

- user interface and application domain code must be separated (a prerequisite for client server) and
- objects are tightly coupled code fragments with persistent data.

The second phase has to compute small but correct interfaces. Both phases work on control and flow-graphs. The first phase uses graphical methods, while the second phase must preserve semantics.

A comparison with our approach shows, that we stress the first point, the second is only shortly discussed as

- semantics is a strength of Petri nets
- there are a lot of existing graph-based approaches we could use within a framework.

There are many more frameworks on the coding levels but a recent article [Men97] counts only two architecture-recovery environments. Both look in the AST for architectural recognisers such as shared files or task control. This is most useful if computers and domain experts collaborate to investigate the architecture of a piece of software. The human expert can detect for the first time a recogniser and let the tool look for other instances. If the result is not as expected, she or he can either change the definition or try another recogniser.

[Bou74] reports, that they have had to postpone (a euphemism for cancel?) an innovative software reengineering tools workshop, because only two tool producers wanted to participate. Today you find a lot of tools, promising to fix year 2000 related problems, but most of these tools are very specific. The author's experience is that if your job needs a simple but slightly different kind of transformation, you are left alone with your own ad hoc tools.

2.4 Different Reverse-Engineering Techniques

A common clustering-based method is to partition a system into high cohesion, low adhesion subsystems. This directly reformulates as a graph theoretical problem. An elementary overview is given in [Wig97].

Clustering can use very different definitions of near. For example [Sif97] selects a set of attributes (from variable usage, type selection etc.) and calls a set of Functions F a concept iff

$F = \{f' \mid f'$ is a function of the system having all attributes in A

with A the set of attributes common to all $f \in F\}$

These concepts form a partial order and one can select a hierarchy of modules. This is in a very abstract sense similar to our folding approach, in that it groups elements, having similar connections. The definition of connection is in both approaches very open, but processing of these connections uses different concepts.

Normally clustering works on nodes, edges are reflected as couplings or attributes between two nodes. In our approach edges are first class citizens, in that they define neighbourhood and allowed/disallowed mergers are defined by edges. An approach that really deals with edges is presented in [Man96]. It defines tube graphs and formal criteria, which edges to merge.

Cliché recognition tries to identify programming constructs in existing code. Typically these techniques – refer for example [Ree97] or [Pal97] - use an program representation that lays somewhere between abstract syntax trees, flow/control graphs and string representations. The specific mix of representations and algorithms allows detecting different clichés or efficiency in a different situation. For example [Fiu96] uses clichés to recognise typical architectural connectors like shared files or task control.

[Sel98] (and [Bra97] in an earlier version) uses patterns to detect (and transform) features in programs for example year 2000 errors. They always speak about context free grammars but they are using variable bindings. Thus, their patterns (and transformation) remind patterns used in logic programming. The big advantage is that the patterns can be formulated natively, e.g. in a syntax consistent with the specific language. The hope is that this consistency drastically reduces the acceptance problems in the industry. To reduce variations, pre-processing standardises the program source. Two points are interesting to us

- pre-processing corresponds to the translation to a Petri net
- using simpler techniques, in form already familiar to the user

In [Deu97], [Qui97] and [Woo98] reformulate plan base program understanding as a constraint satisfaction problem (CSP) and try to solve it with CSP techniques. They finally are successful in recognising a very elementary plan in programs with several thousands lines of code – which they claim to be an improvement of an order of magnitude over the state of the art. This research has several interesting aspects:

- The modification at the CSP engine can been seen as re-introducing the graph search techniques (essentially in the data flow graph of the program to analyse). A possible conclusion: currently graph-based techniques can't be avoided.
- Whereas the tractable size of the program seems to approach reasonable dimensions, this is not true for the size of the plan library. The techniques are restricted to one very elementary plan. [Woo98] presented a hierarchical algorithm. But the only experiment he could do with it showed higher cost than the flat algorithm.
- Not much experience is available for a cliché library applicable to real world programs and reverse-engineering tasks. It seems that a basic dilemma is not solved. For usability the library should contain few general plans. But for recognition efficiency, a plan must have unique features.
- General plan recognition techniques use open perception assumption. I.e. if an observation for a plan is missing they do not reject the plan. But for program analysis this is wrong. If a piece of code is missing it is definitely not in the code. If something is incomplete it is the plan library not the code.

Our conclusion is that AI systems which rely on a deep understanding of the software are not yet ready for real world reverse-engineering tasks. This could be a sign that the rationalistic perspective [WF86] is not fully appropriate for software reverse engineering. This is quite surprising, as nothing seems wrong to consider the software artefacts as

- closed symbolic virtual worlds
- designed and implemented within the rationalistic perspective

It is out of the scope of this paper to discuss which metaphors of new AI could preferably be applied to reverse engineering. But the immediate consequence for our work is to experiment (also) with other metaphors. The Petri net metaphor explored in this work of course can be interpreted as another tool within the rationalistic perspective (even within formal methods). But we can also interpret for example as a metaphor for situated design: An engineer sitting at a terminal, reading code and drawing diagrams.

Reverse engineering may use other inputs than source code. Dynamic trace data collected during the execution of the system may reveal information that may not be easily from the static program source.

[Wil96] instruments source code with a dynamic trace facility. By comparing traces from runs of different test cases, they get hints, where to look for the implementation of certain features. The basic technique is to run a test case without the interesting feature and one with this feature. Code traced only by one of the two tests has a closer or looser relationship with the searched functionality. Our work does some fundament generalisations on that

- We do not need the source code. We can trace at an interface for example to a database.
- Depending on the application we also can use code locations, but we may get a different result: a high-level model of the software investigated.

2.5 Flexibility and 'Take it Easy"

Many proposals for a flexible reverse-engineering strategy exist. Most of them involve human experts as pilots. Such frameworks can be used as a tool for general maintenance. But they are also useful for forward engineering because things which are easy in a single-language environment (e.g. *show all senders* in Smalltalk) get awfully complex in a heterogeneous environment. Refer for example to [Kul98] for a situation on a MVS system. The described situation is not extreme – the author knows worse ones.

[Til93] describes flexibility as one of the three main design goals of Rigi (a framework for discovering and analysing the structure of large software systems. Named after a mountain in central Switzerland, known for it's easy access by two of the very early mountain railways). Flexibility is considered in at least three dimensions:

- parsers for different languages
- extensible and configurable analysis algorithms
- tailorable views and user interfaces

In [Kaz98] a framework for reverse and re-engineering tools is presented. He uses a horse shoe metaphor shown in Figure 2. The idea is the following sequence:

- reverse engineer a legacy application from source code up to higher levels
- do an architectural transformation
- Forward engineer to get the desired new implementation.

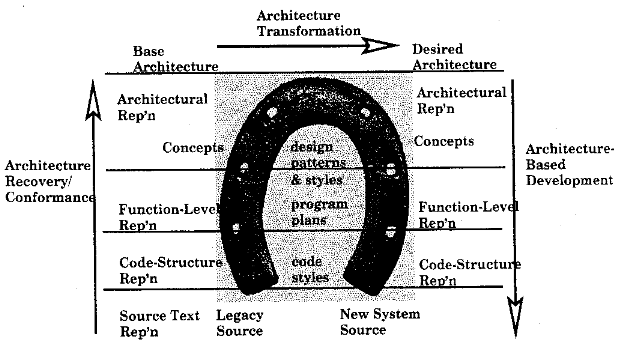


Figure 2. The horse shoe.

[Ant97] presents the maintenance Environment CANTO. The user communication consists of a program source editor and a graph display. For each user action they show the results of the requested analysis by different tools on the architectural and the program logic level. The tools work on an intermediate representation extracted by a front end. The major benefit is not a single analysis by itself. It is the seamless interface which allows to navigate intuitively between different representations and to modify the system in the same environment. Analysis on the architectural level is also done with help of user expertise.

Reverse engineering faces many difficulties. But there is also another side: Very simple techniques can give surprising results.

[Kon97] experiments with different combinations of program metrics to detect cloning in programs. The detection quality is measured by methods commonly used in information retrieval. A similar work is [May96] using different metrics and a different point of view. One interesting point for us is, that they use software metrics which is a tool which is - for the current purpose - rather superficial. Superficial compared to formal methods which try to reflect the precise semantics of a program. Also, superficial compared to heuristics which mimic human thinking about software. But this superficial approach has advantages in performance, tolerance for programming errors etc.. The superficiality may work as a fuzzy filter – which is difficult for formal methods. In fact these methods could claim to detect program structure, not only clones, just by changing the point of view. The difference between *two procedures are clones* and *they are instances of the same pattern* is only gradual.

We can see some similarities to a possible application of our approach: we take a program and draw a Petri net from it, for example by simple visual methods. And we get something software engineers are working with every day: diagrams that give a high-level conception of a software system. These diagrams are correct only in an informal or fuzzy way.

Interestingly enough, an opposite application of superficiality is useful. In [Bak95] two sections of program text match iff they are equal

- after removing comments
- squashing multiple spaces, but preserving the line structure
- and global substitution

An algorithm detecting such matches is already a powerful tool to detect cloning. [Gup97] uses even simpler patterns. They are restricted to a single line of source. We can interpret these techniques (after possibly adding some constraints), as applying formal semantic preserving transformations. But these transformations are selected not by a semantic criterion, but from superficial properties like simplicity, peculiarities of the used programming language and programming practise and efficiency of the analysis algorithm.

2.6 Nets in Reengineering

Petri nets are rarely used for reverse reengineering. This is rather surprising, as many different kinds of graphs are used or invented in software maintenance like different forms of dependence trees. In [Cre97] a graph rewriting system is used as the transformation engine for reverse engineering. Hence graphs techniques can also be used as transformation rules – not only as static descriptions.

But Petri nets are more often used to model the software engineering process or business reengineering. For an example refer to [Avr95] or [Kel95]. But, to the authors knowledge, no such tool contains a reverse-engineering component.

[Chu96] translates formal components to Predicate/Transition nets. A formal component here is specified in a specification language called MIATAC-MA, by three sets of logic formulas:

- pre-conditions: must be satisfied in order to use a component correctly
- post-conditions: are satisfied after correct use of a component
- obligations: tasks that have to be done sometimes in the future (i.e. closing a file after opening it).

The predicate transition net contains a single input place with the preconditions, one output for the post-conditions and another one for the obligations. This translation can be used for our method, as a further variant to get a net. But in our opinion the title of [Chu96] is misleading, we cannot see any reverse- or reengineering done in this paper.

In [Can93] not a net but a bipartite graph is mentioned. Within RE^2 project reuse candidates are searched by different clustering measures in a bipartite graph. One kind of nodes corresponds to procedures, the other kind to global variables and the edges to variable references, that always run from procedure to variables.

3 Petri Nets

3.1 Introduction

This chapter is the net-theoretical part of this thesis. As discussed in sections 1.1 and 2.1 there exist Petri-net classes from the literature fulfilling some of our requirements, but, new definitions are needed to fulfil the combination of all requirements.

We decided to use the framework of category theory which is a mathematical meta-theory such as logic. Category theory stresses

- relationships between nets and
- relationships between net classes

rather than single transitions or places of a net. This is very natural for forward and reverse engineering. Basic definitions of category theory and our motivations for the use of categories in this work are detailed in the following:

Remark 1. Category theory in computer science. ⇨102◈

The balance of this chapter is as follows. The following section introduces some notations and defines the basic categories. In section 3.3 clustering-based Petri-nets are introduced. Then a bridge to folding-based Petri-nets is created. Later it is demonstrated, that these Petri nets are a sound basis for defining high-level nets, additionally universal constructions with applications in reverse engineering are presented. Finally, the remaining section describes the integration of behaviour and semantics in the current categorical framework.

3.2 Preliminaries

This section introduces some notations, reviews some well-known categories and introduces the category of one-sets. This is new variant of multisets avoiding some their disadvantages.

3.2.1 Notation

For a general introduction to Petri nets refer to [Rei98] for category theory to [Lan71]. Used symbols, abbreviations and terms are listed in chapter 9 Index and an introduction to Smalltalk is contained in Remark 67. We omit the description of symbols that are usual. To avoid confusions we provide the following list:

$f\colon X \to Y$: A function, a partial function, a morphism, a functor or a natural transformation from X to Y

g f x: Multiplication, function composition and functor application may be written with or without operator and/or brackets. If clear from the context functions are lifted to multisets or power sets without special notation.

undef: f x = undef means the partial function f is not defined on the element x

$\underline{C}[X, Y]$: the set of morphisms from object X to object Y in the category $\underline{C}$

$[X, Y]$ is $\underline{C}[X, Y]$ if the category is understood

Two functor arrows connected by a triangle (Figure 3 top) symbolise an adjunction. The triangle points from the left adjoint L to the right adjoint R. As shown at the bottom of Figure 3 this is also the direction of unit and counit and morphisms which are transferred by the natural equivalence η.

For a coreflection the adjunction symbol is decorated with a vertical equal sign at the side of the unit which consists of all isomorphisms (Figure 3 left middle) Similarly a reflection gets an equal sign at the side of the counit.

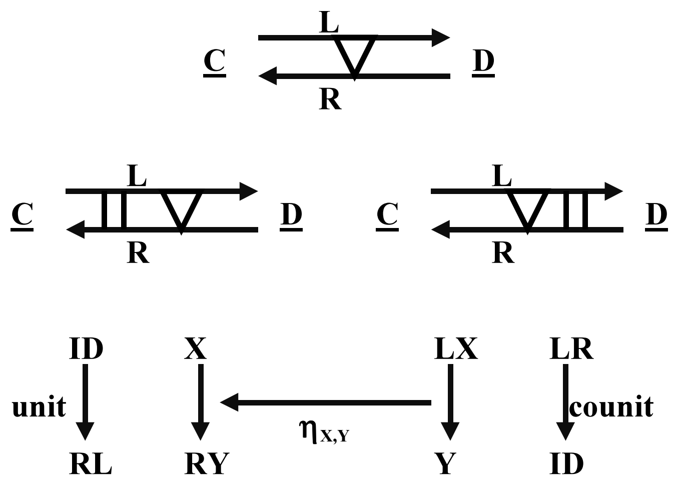


Figure 3. Symbols for adjunction (top) and (co)reflection (middle) and mnemonics (bottom).

3.2.2 Sets and Multisets

Definition 2. <u>SETP</u> designates the category of sets and partial maps. As usual the composition of partial maps is defined by

(g f) x = if f x = undef then undef else if g (f x) = undef then undef else g (f x)

It is well-known that in <u>SETP</u> all (co)universal constructions exist. As it is important to understand what these look like, the reader might like to consult the proof of

Proposition 3. <u>SETP</u> is cocomplete and complete. ⇨104◈

Definition 4. A multiset over a set S is a function $m \in MS(S) = \{f: S \to \mathbb{N} \mid f(s) \neq 0\}$ for a set S. The sum of two multisets is defined by $(m'+m'')(s) = m'(s) + m''(s)$ and similarly multiplication with a natural number. The category <u>MS</u> consists of the objects MS(S) for a set S and morphisms which are the linear functions $MS(S) \to MS(S')$. MS extends in the obvious way to a functor from <u>SETP</u> to <u>MS</u>.

As usual, linear means $f(m'+m'') = f(m') + f(m'')$ and $f(\lambda m) = \lambda f(m)$ the latter being an obvious consequence of the former. **0** is the empty multiset ($\mathbf{0}(s) = 0\ \forall s \in S$). We freely switch between the function notation (m(q) is the multiplicity of q in m) and formal sums ($m = 3s' + s''$). Further we define $m' \leq m''$ iff $m'(s) \leq m''(s)$ for all $s \in S$.

The last section showed that <u>SETP</u> is finitely complete and cocomplete. This is not true for <u>MS</u>:

Lemma 5. In <u>MS</u> there are finite diagrams without a colimit or a limit. ⇨106◈

The counter examples in the proof of the above lemma are very simple. It signifies that universal constructions are missing in typical situations in <u>MS</u>.

This is an uncomfortable dilemma for the construction of Petri net categories. Multisets are widely accepted to model resources – but they lack universal constructions. Unfortunately the obvious generalisation to $\mathbb{Z}$ -modules that would allow universal constructions is lacking the modelling power. Over $\mathbb{Z}$ +r and –r have equal rights. But for nets the difference between an available resource +r and a missing resource –r is crucial. Hiding this difference in a tiny sign compromises the essence of net theory.

A simple way out is to only use 1-dimensional multisets or simply the naturals. But this is not a real solution:

- the distinguishing feature of high-level nets is, that their places carry not only a number but also something more complex, normally a kind of multiset.
- even if every place only carries a number, a marking is still a multiset. At least as long as nets are not restricted to a single place – which would simplify net theory a little bit...

Another way to handle the dilemma is a restriction of the type of morphisms used. In the next section we construct a subcategory of <u>MS</u> having the same objects but only a subset of the morphisms. The category has nice adjunctions to <u>SETP</u> from which we will profit in the Petri net categories.

3.2.3 One-Sets

The category <u>1S</u> is the result of careful balancing of different requirements.

- it is a subcategory of multisets
- it has all colimits and limits
- morphisms retract to the base sets
- it allows infinite base sets

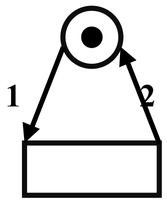


Figure 4. A net with an infinite reachability graph.

The last point does not seem necessary for a computer tool. But, the behaviour of a net can be modelled by another net - a nice self-reflecting feature of net theory. However for a net as small as the one in Figure 4 the semantics might create an infinite net.

The new category is connected by three different functors with <u>SETP</u> – two of them forming an adjunction. The adjunction clearly shows that <u>1S</u> is smaller than <u>MS</u> because the right adjoint functor maps a one-set to a set much smaller than the set of elements.

The section finishes by showing that <u>1S</u> is cocomplete and complete. In the next chapter the definition of nets will be very concise because we have all necessary categories at our disposal.

Definition 6. <u>1S</u>, called one-sets, is the subcategory of <u>MS</u> with multisets over a base set as objects. A morphism f∈<u>1S</u>[M, M'] is the linear expansion of the product of two functions, namely the base component f_β and the coefficients f_γ with:

$f(\sum_{s\in S}\lambda_s s) = \sum_{s\in S}\lambda_s (f_\gamma s)(f_\beta s)$ for $\lambda_s \in \mathbb{N}$ such that

$f_\beta \in$ <u>SETP</u>[S, S']

$f_\gamma: S \to \mathbb{N}$

def $f_\beta = \{s \in S \mid f_\gamma s \neq 0\}$

S the base set of M

S' the base set of M'

In other words <u>1S</u> is a subcategory of <u>MS</u> with the same objects and the restriction on morphisms that a base element is mapped to the multiple of a base element. This is a generalisation over partial functions but a specialisation from multisets morphisms. It is easy to see that the morphisms are compositional. A 1S morphism determines f_β and f_γ. Reverse if $\text{def}(f_\beta) = \{s \mid f_\gamma(s) \neq 0)$ then they uniquely determine an f. $f = (f_\beta, f_\gamma)$ denotes this situation that allows layered constructions.

Definition 7. 1S: SETP$\rightarrow$1S is the functor mapping a set S to the one-set with base S and a partial function f to a 1S morphism with

$\forall s \in S$: (1S f) s = if $s \in$ def f then 1 (f s) else **0**

(1 (f s) is f s interpreted as a multiset).

This functor builds the one-set over a given set. It is well-defined and compositional because the 1S morphisms with coefficients 1 are a direct translation of partial functions on the base sets.

Definition 8. B: 1S$\rightarrow$SETP is the functor that maps an object of 1S to its base set (i.e. B (1S S) = S) and retracts a morphism f to f_β.

B is the inverse to 1S on objects by mapping a one-set to its base set (for morphisms see Proposition 10). This is possible for 1S-morphisms but not in general for MS-morphisms. Compositionality follows because undefined values propagate in partial function composition in the same way as zeros do in products.

Definition 9. BN: 1S$\rightarrow$SETP is the functor mapping a object M to the set $\mathbb{N}^+$ x (B M) and a morphism $f \in$ 1S[M, M'] to

def (BN f) = $\mathbb{N}^+$ x (def f_β)

(BN f) (s, λ) = $\lambda\, f_\gamma(s)\, f_\beta(s)$ for $s \in S$ and $\lambda \in \mathbb{N}^+$.

BN maps a one-set to the base skeleton, which is the set of nonzero multiples of base elements. By the same argument as above BN is well defined. The three functors are related as follows:

Proposition 10. 1S is left adjoint to BN. There is a natural equivalence ε: $\text{Id}_{\text{SETP}} \rightarrow$B 1S and there are isomorphisms δ_M: M$\rightarrow$1S B M with B $\delta_M = \varepsilon_{BM}$. ⇨107◈

$$\begin{tikzcd}
\text{SETP} \arrow[r, shift left=1.5, "1S"] \arrow[r, phantom, "\vee"] & \text{1S} \arrow[l, shift left=1.5, "BN"]
\end{tikzcd}$$

Proposition 11. 1S is cocomplete and complete. ⇨108◈

The computation of a colimit is computationally reasonable – the size of the coproduct does not exceed the size of the disjoint union, that is the size of the input. But this is not true for limits. Already the product of one-sets with the base consisting of a single element gets an infinite base. Although 1S is complete products are only of restricted computational value.

In summarising this chapter we compare 1S and MS. For both categories the free functor from SETP has a right adjoint. For a multiset this is the usual forgetful functor producing the set of all elements of the multiset, for a one-set it is the smaller set of the base skeleton. Furthermore the functor 1S has a left inverse functor which is simply B. And finally 1S is cocomplete and complete which isn't true for MS – worse, simple diagrams typically are lacking a colimit or limit.

3.3 Place-Transition Nets

We start with one side of the framework: a class of Petri nets with clustering that will be called PTNET: the category of place-transition nets.

Clustering should support software-engineering principles for subsystem composition. The well-known graphical representation of Petri nets should be more than a mere visualisation. Graphical operations should be interpreted in a natural and intuitive way within PTNET. And not least, it should be a real Petri-net category in the sense that it reflects Petri-net principles and can harness the power of Petri-net analyses.

Although there are definitions in the literature with different combinations and variations of these properties, we need the new PTNET category to be able to link to a folding category later on.

Whereas PTNET, the category of place-transition nets, defines nets (the objects of the category) in a standard way, morphisms are a novel combination of:

- vicinity-preserving graph morphisms
- foldings commuting with the pre and post function, and hence mapping transition occurrences

Such a morphism retracts to the underlying graph. If a morphism is seen as an implementation/specification relation, then the origins of a node are a subsystem of the source net. The origins of a transition form a transition-bordered subnet with proper port transitions. Ignoring vanishing nodes, the origins of a place form a place-bordered subnet. Hence PTNET morphisms are vicinity-preserving and support clustering in a way which is very useful for composing subsystems.

A morphism retracts to a (partial) graph morphism. Thus PTNET has a direct interpretation of operations on the underlying graph. This is an important principle of Petri-net theory - the graph is not a mere visualisation but a major component. Thus graph operations and transformations are a natural and powerful tool in PTNET. This claim is supported by a coreflection between 1S and PTNET.

Place and transition invariants are important structural net properties. A morphism transfers a place invariant from the destination net to the source net. This leads to a very convincing interpretation of semi-positive place invariants: they are simply morphisms to a net consisting of a single place. A transition invariant is transferred in the direction of the morphism. This shows that PTNET integrates smoothly with linear algebraic techniques which are important net-theoretical tools.

3.3.1 Definitions and Basic Properties

Definition 12. The category PTNET of place-transition nets consists of objects

$N = (\mathrm{pre}_N, \mathrm{post}_N \in \underline{\mathrm{MS}}[\mathrm{MS}\ T_N, \mathrm{MS}\ P_N])$

with disjoint sets T_N of transitions and P_N of places.

$X_N = T_N \cup P_N$

is the set of nodes of the net N and f is a morphism $N \rightarrow N'$ iff

$f \in \underline{1S}[1S\ X_N, 1S\ X_{N'}]$,

$\forall t \in T_N$: (f_β $t \in T_N$' and f pre_N t = pre_N' f t) or f pre_N t = f t and
$\forall t \in T_N$: (f_β $t \in T_N$' and f $post_N$ t = $post_N$' f t) or f $post_N$ t = f t
A morphism f is called

- a folding iff f_β P $\subseteq$ P' and f_β T$\subseteq$T'
- unitary iff f_γ $X_N \subseteq \{1\}$ and
- binary iff f_γ $X_N \subseteq \{0, 1\}$ ⇨109◈

This means that the pre and post functions map a transition to a multiset of places. A morphism maps a node of the source net to a multiple of a node of the destination net and the pre (or post) multiset of a transition is either mapped to the image of the transition or its pre (post) multiset. It is worth mentioning that a node may be mapped to **0** and that the empty net is the zero object of the category.

These morphisms are a combination of commutative and vicinity-respecting definitions. Before the *"or"* in the definition stands the usual commutative requirement and after it a strengthened version of vicinity-respecting. Vicinity-respecting morphisms normally are graph based hence (partial) functions from X to X'. Commutative definitions map places to markings but transitions to transitions - an asymmetry that becomes problematic for the clustering of places with transitions. We keep the middle with 1S morphisms - a restricted form of multiset morphisms that retracted to a (partial) node function.

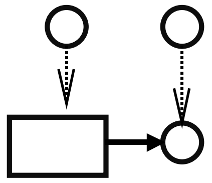


Figure 5. Mono and epi but not iso.

As usual we will replace the index N with other forms of sub- or superscripting or drop it completely if the context rules out misunderstandings. We will use the convention that the application of a pre or post function implies the argument to be a nonzero transition multiset. Thus the conditions before the *"and"* become superfluous.

The objects are defined in the standard way the notation is similar to [NS98]. Many usual notations are easily derived, for instance:
the pre-set of a node: ${}^\bullet x = \{y \in X \mid \text{if } x \in T \text{ then } pre(x)(y) \neq 0 \text{ else } post(y)(x) \neq 0\}$
the postset of a node: $x^\bullet = \{y \in X \mid \text{if } x \in T \text{ then } post(x)(y) \neq 0 \text{ else } post(y)(x) \neq 0\}$
the neighbourhood of a node: ${}^\bullet x^\bullet = {}^\bullet x \cup x^\bullet$
the environment of a node: $env(x) = \{x\} \cup {}^\bullet x^\bullet$
the flow relation of the net: $F = \{(x, y) \mid x, y \in X \text{ with } x \in {}^\bullet y\}$

The basic morphism types are characterised by:

Proposition 13. A PTNET morphism f: N→N' is

- epimorphic iff f_β is surjective
- monomorphic iff def(f_β) = X and f_β injective
- isomorphic iff f is unitary, f_β is a bijective and isolated places are mapped to places.

⇨109◈

Figure 5 shows that in PTNET monomorphic and epimorphic does not imply isomorphic. Not even for finite nets. A morphism such as this is useful in engineering:

- it does not loose any information because it is mono

- it does not 'invent' anything new because it is epi
- it transforms information because it is not iso.

Remark. The proposition still holds in subcategories of PTNET allowing only foldings (FNET) or disallowing morphisms to map places to transitions (PPNET).

An isomorphism in PTNET is a bijective unitary folding or equivalently a pair of bijections, one between the places and one between the transitions. Hence our Petri net isomorphisms are the exactly the same as in clustering-based ([GLT79]) or folding-based ([NS98]) approaches. On the other hand, even our foldings are in general no morphisms of [NS98] because they are allowed to multiply transitions.

The tight relationship from Petri nets to the underlying graph is first seen by the definition of a PTNET morphism as (special) 1S morphism. This allows retracting a net morphism f to f_β. But there is also something to say about the other direction:

Definition 14. PTNET: 1S→PTNET is the functor with PTNET 1S P equals the net with places P and no transitions and the identity on morphisms. U is the underlying functor sending a net N to 1S X_N.

Proposition 15. PTNET: 1S→PTNET is the left adjoint of U. Moreover, they form a coreflection. ⇨110◈

$$\begin{tikzcd}
\mathbf{SETP} \arrow[r, shift left=1.5, "\mathbf{1S}"] \arrow[r, phantom, "\nabla"] & \mathbf{1S} \arrow[l, shift left=1.5, "\mathbf{BN}"] \arrow[r, shift left=1.5, "\mathbf{PTNET}"] \arrow[r, phantom, "\parallel\ \nabla"] & \mathbf{PTNET} \arrow[l, shift left=1.5, "\mathbf{U}"]
\end{tikzcd}$$

Observe that 1S and PTNET compose having a right adjoint BN U. On the other hand B U f equals f_β the retraction of a morphism to the underlying graph.

3.3.2 Clustering

A simple way to use morphisms for clustering is to interpret a morphism say f as a specification of an implementation. The source of f is the implementation and the destination is the design. The origins of a destination node form a subnet of the source. This of course works for any function between the node sets. But crucial are the specific properties such subnets have.

The idea that the origins of a node form a super node is not too bad as Figure 6 shows. Moreover, a closer look moreover reveals that super transitions have special input and output transitions. This is formalised in the following:

Definition 16. Let S, R$\subseteq X_N$. S is called non-splitting relative R iff for each transition $t \in S \cap T$ the intersection ${}^\bullet t \cap R$ is either completely contained in S or disjoint from it and similarly for $t^\bullet$. S is called non-splitting iff it is non-splitting relative X_N

Proposition 17. Let f: N→N', $K_f = X_N \setminus def(f_\beta)$ and S'⊆X'. Then

- $def(f_\beta)$ is transition-bordered
- K_f is place-bordered,
- if S' is place-bordered then also $f_\beta^{-1}(S') \cup K_f$
- if S' is transition-bordered then also $f_\beta^{-1}(S')$
- if S' is non-splitting then also $f_\beta^{-1}(S')$ relative $def(f_\beta)$

⇨111◈

Ignoring K_f - the 'garbage component' - the origins of a single node form a super node of the same type. Furthermore, super transitions have proper port transitions. These clustering capabilities of PTNET morphisms have obvious parallels to modularization principles in software engineering. As well as having strong parallels to several classes of Petri nets with composition operators.

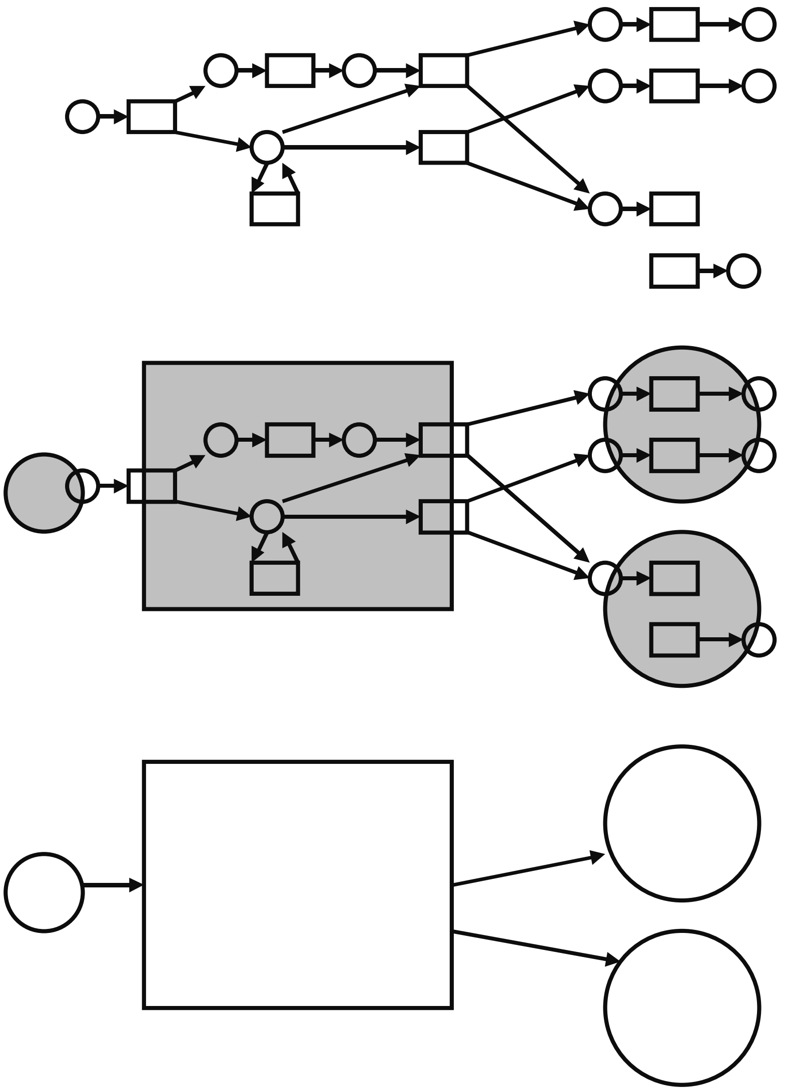


Figure 6. The origins of a node form a super node.

What about the reverse? Given a set S of nodes, is there a morphism that exactly collapses these nodes? If S consists only of places the obvious map from X→X\S∪{S} yields a unitary folding. But for a set of transitions in general this is not true. They need to be compatible. The relationship between morphisms and net invariants elaborated in the next section allows the formulation of a linear algebraic criterion whether such a collapsing morphism for an arbitrary S⊆X exists.

3.3.3 Net Invariants

Place and transition invariants are important tools in net theory. They allow deducing behavioural properties by linear algebraic techniques from the net structure. A place invariant is a map from markings to the integers that does not change under transition occurrence. A transition invariant is a transition sequence with total marking change zero:

Definition 18. A place invariant of a net is a linear function i: 1S P→ ℤ with i (post - pre) = **0** and a transition invariant j is an integer linear combination of transitions with (post - pre) j = **0**. An invariant is called semi -positive iff all coefficients are non-negative.

Proposition 19. Let f: N→N' a morphism of place-transition nets. Then

- if i': X_N'→ℤ is a place invariant of N' then $i = \sum_{p \in P} i' f p$ is a place invariant of N.
- Semi-positive place invariants of N are 1 to 1 with morphisms to single place nets.
- if j∈1S T_N is a transition invariant of N then $j' = \sum_{\substack{t \in T \\ f\,\mathrm{pre}\,t \neq f\,t}} j(t)\,f(t) = \sum_{\substack{t \in T \\ f\,\mathrm{post}\,t \neq f\,t}} j(t)\,f(t)$ is a transition invariant of N'.
- A semi-positive transition invariant corresponds to a unitary folding from a T-system (a PTNET with all arc cardinalities 1 and $|{}^{\bullet}p| = 1 = |p^{\bullet}|$ for all places).

⇨111◈

A semi-positive place invariant has a very concise characterisation: it is simply a morphism to a single place net. A transition invariant corresponds to a T-system which is neither unique nor small. The asymmetry between place and transition invariants is a consequence of the asymmetry in their definitions as well as the break of the place/transition duality by the definition of a PTNET morphism.

Remark 20. The major limitations of PTNET are the following: universal constructions do not exist in general and behaviour transfer is complicated. ⇨113◈

3.4 From Clustering to Folding

The folding side of our framework will be called FNET, the category of nets with foldings. The crucial point is to obtain a powerful relationship between the two sides, FNET and PTNET. A one-way relationship such as a functor (e.g. subcategory) is not enough. Simulations in both directions are needed. We will use a strong relationship of this kind which is formalised in category theory under the name of adjoint functors. An adjunction is the first choice from a theoretical point of view but computational complexity must be affordable for practical applications. If a subnet is mapped to a transition, its 'occurrences' are no longer atomic like transition occurrences. There are several propositions in the literature for dealing with such transitions in occurrence - but they add complexity and violate the principle of transition-occurrence atomicity. A solution within pure Petri-net theory would be preferable.

The category representing folding is the subcategory of PTNET containing the same objects but in which only foldings are allowed as morphisms. Furthermore, a middle category of place-preserving nets to be known as PPNET is defined. Its morphisms map places to places.

The bridge from clustering to folding is formed by two adjunctions which compose at PPNET. They allow a place-transition net to be simulated by a place-preserving or a folding net - in a faithful and computationally reasonable way. Moreover, they also show how to handle 'transitions in occurrence' in an elegant way while remaining within pure Petri-net theory.

On a theoretical level, these adjunctions clarify the relationships between two groups of Petri-net formalisms:

- vicinity-preserving or graph-based
- algebraic or behaviour-preserving.

FNET and PPNET have further interesting properties in addition to those of PTNET. For example, all couniversal and universal constructions exist. This allows nets to be composed from subnets in a flexible and natural way. It also allows graph-rewriting techniques to be applied directly on the basis of the single or double-pushout approach.

3.4.1 Place-Preserving Nets

The category of place-transitions nets offers powerful features for clustering based software engineering and for structural net analyses. It's weaknesses are the lack of universal constructions and the difficulty to transfer behaviour. A small modification improves the situation:

Definition 21. A PTNET morphism $f: N \rightarrow N'$ is called place-preserving iff $f_\beta(P) \subseteq P'$. The category PPNET is the subcategory of PTNET with the same objects and place-preserving morphisms.

Obviously pp stands for place-preserving. The following lemma gives it yet another sense:

Lemma 22. Let $\Delta x = (x,x)$ and $pp = (pre, post) \in \underline{MS}(MS\ T, MS\ P) \times \underline{MS}(MS\ T, MS\ P) \cong \underline{MS}(MS\ T, MS\ P \times MS\ P)$. For a PTNET morphism $f: N \rightarrow N'$ the following properties are equivalent:

- f is place-preserving
- $(\forall t \in T$: $f\ pp\ t = pp'\ f\ t$ or $f\ pp\ t = \Delta\ f\ t)$ and
  $(\forall p \in P \cap def\ f_\beta$: $\exists p' \in P$ with $f_\beta\ p' \in P'$ such that p and p' are connected by an undirected path with all nodes in $def(f_\beta))$.

⇨114◈

Thus the second understanding for pp is pre and post. Arguments for pre and post may be replaced by an argument with the combined function pp. This abbreviates notation and simplifies proofs. For example we may abbreviate a net N = (pre, post) by N = pp.

Clearly there is a forgetful functor U: PPNET→PTNET simply forgetting the place-preserving restriction on morphisms.

Proposition 23. The forgetful functor U has a right adjoint PP: PTNET→PPNET. ⇨114◈

$$\begin{tikzcd} \underline{\textbf{PPNET}} \arrow[r, shift left, "\mathbf{U}"] \arrow[r, phantom, "\nabla"] & \underline{\textbf{PTNET}} \arrow[l, shift left, "\mathbf{PP}"] \end{tikzcd}$$

PP replaces every transition by three transitions and a place as shown in Figure 7. This deals with the four combinations of the image of the pre- and postsets of a transition under a PTNET morphism. Figure 7 suggests an upward projection which merges again the 4 nodes generated from each transition. This projection is a morphism and moreover the counit of the adjunction.

A place-transition net can be faithfully simulated by a place-preserving net. This is computationally sensible, as the place-preserving net gets 3 additional nodes for every transition in the original net. Given the simplifications in the new category this seems no bad programming technique.

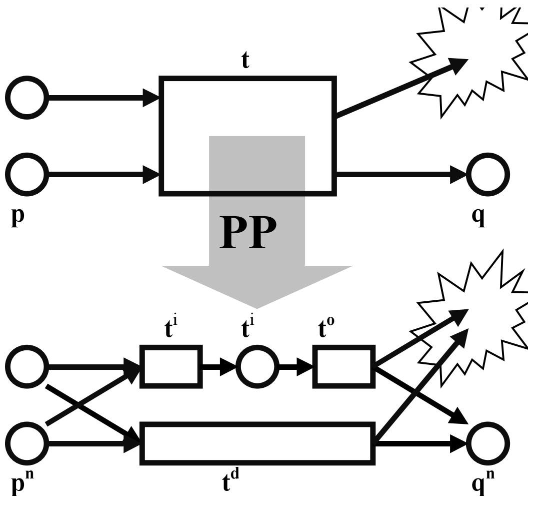
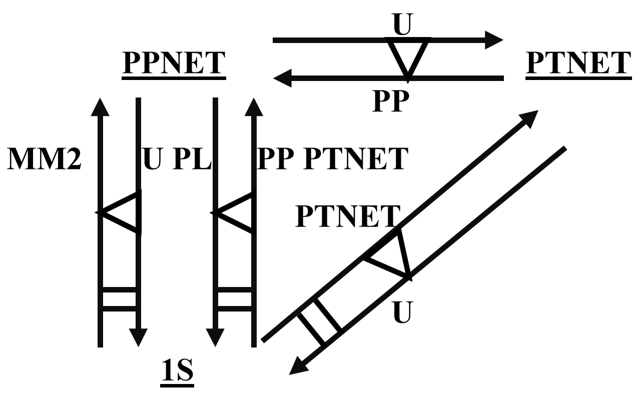


Figure 7. The Functor PP.

Observe that the two adjunctions between PTNET and 1S respectively PPNET do not compose as adjunctions because left and right do not match. However there are other adjunctions:

Proposition 24. Let PL: PPNET→PPNET be the functor dropping all transitions and keeping the places of a net. U PL: PPNET→1S forms a coreflection with the left adjoint PP PTNET and forms a reflection with the right adjoint MM2: 1S→PPNET. ➪115◈



The diagram beside summarises the situation. Notice that it is not commutative, e.g. U U ≠ U PL and the cyclic triangles compose to identities only in special cases (e.g. U PL PP PTNET = Id_{1S}) but not in general. There are two adjunctions between PPNET and 1S although the adjunctions between PPNET and PTNET and between PTNET and 1S do not compose.

Proposition 25. PPNET is cocomplete and complete. ➪116◈

U PL is a left adjoint and hence transfers limits. Hence the places of a limit are determined by the limit in 1S but the limit may 'invent' additional transitions required by commutativity with pre and post. If U U would transfer limits it would disallow this and hence limits could not exist in general. This 'invention' of transitions may produce problems with computational complexity. This cannot happen with colimits: their size is limited by disjoint union and hence by the size of the input.

3.4.2 Folding Nets

Definition 26. FNET is the subcategory of PPNET with the same objects but only foldings as morphisms.

Figure 8 shows different mappings of nodes and tabulates in which categories they are allowed for morphisms. It lists our three categories, a typical clustering-based one ([GLT79]) and a folding-based one ([Win84a]). For a more formal comparision see Figure 20.

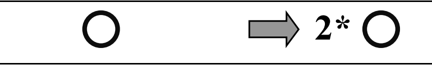
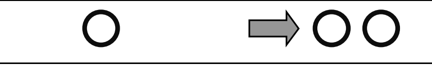
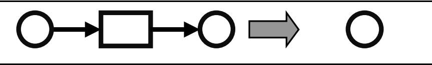
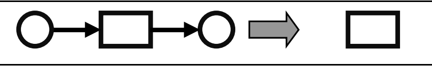
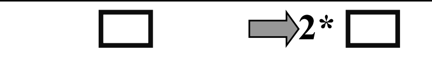
	[GLT79]	PTSYS	PPSYS	FSYS	[Win84a]
	∅	√	√	√	√
	∅	∅	∅	∅	√
	√	√	√	∅	∅
	√	√	∅	∅	∅
	∅	√	√	√	∅

Figure 8. Allowed (√) and disallowed (∅) mappings for different morphisms.

Figure 9 demonstrates how to define a functor F: <u>PPNET</u>→<u>FNET</u>. Whereas PP splits transitions, F splits places. F has similar properties as PP and the two adjunctions compose:

Proposition 27. The underlying functor U has a right adjoint F: <u>PPNET</u>→<u>FNET</u>. ⇨118◈

$$\begin{tikzcd}\underline{\mathbf{FNET}} \arrow[r, shift left, "\mathbf{U}"] \arrow[r, phantom, "\nabla"] & \underline{\mathbf{PPNET}} \arrow[l, shift left, "\mathbf{F}"] \arrow[r, shift left, "\mathbf{U}"] \arrow[r, phantom, "\nabla"] & \underline{\mathbf{PPNET}} \arrow[l, shift left, "\mathbf{PP}"]\end{tikzcd}$$

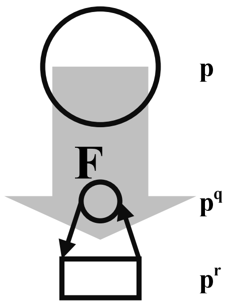


Figure 9. The Functor F.

Proposition 28. <u>FNET</u> is cocomplete and complete. ⇨118◈

Later we will need morphisms that map transitions only to similar transitions:

Definition 29. A morphism f is called local injective iff it is a unitary folding with f_β injective on the environment of each transition.

3.5 High-Level Nets

To model real-world applications with the formalism of ordinary Petri nets alone is cumbersome, and is rendered practicable only by the additional expressiveness of high-level nets. High-level nets such as coloured or hierarchical nets provide a compact notation for large (e.g. infinite) nets. The design of such net classes has to balance two conflicting requirements: expressiveness versus coherence. Coherence allows analyses to be performed for an unfolded net within a compact notation. However, many tools lack such coherence, as well as categorical resources such as morphisms for simulation or universal constructions for sophisticated subsystem composition.

Initially, a coloured net is defined as a unitary folding and a morphism as a pair of morphisms of the underlying category forming a commutative square with the two foldings. This means that a colour of a token on a place of the coloured net is unfolded to a place in the unfolded net - which corresponds to the unfolding of a coloured net to a flat net. But the usual definitions delegate the coherence of a coloured net and its unfolding to the typing system (e.g. the permitted expression for arc inscriptions) whereas our coloured categories derive the coherence condition from the definition of a morphism, which may be stronger, for instance by not allowing variable arc cardinalities.

The range from clustering to folding is lifted by this construction to coloured nets yielding the following three categories:

- CPTNET: coloured place-transition nets
- CPPNET: coloured place-preserving nets
- CFNET: coloured folding nets

Each of these coloured categories is connected to the underlying uncoloured category by two adjunctions, more precisely a reflection and a coreflection. Furthermore, co-universal and universal constructions in coloured nets reduce to such constructions in the source and destination nets. Although these categories are very abstract, they encompass the net-theoretical essence of existing classes of algebraic or coloured nets - which are already implemented as computer tools. Our categories have the potential to enrich such net classes or tools by a categorical framework including the integration of clustering and folding, morphisms for simulations and universal constructions for subsystem composition.

More generally, a category of hierarchical nets consists of net diagrams and natural transformations. However, similar and more abstract arguments and properties apply, as for coloured nets. And the clustering capabilities of PTNET or PPNET really prove their usefulness for hierarchical nets.

Once high-level nets are used, the question of how to reverse-engineer existing nets arises. Chapter 4 will show that reverse engineering of Petri nets applies not only to Petri nets but also contributes a novel analysis to the reverse engineering of conventional applications. Within this framework, the task of reverse engineering is: given a net N^f, find a hierarchical net that specifies the design of N^f. The hierarchical net is a diagram consisting of colouring foldings and clustering morphisms. Hence the task reduces to three subtasks: to compute colourings, to compute clusterings and to select and compose a meaningful diagram from them. Here we shall only treat the first subtask. Many methods from traditional engineering may be used for the second one. Practical experience with the first two would be a precondition for the third one.

Hence for a given net N^f we have to compute a colouring morphism $c: N^f \to N^u$. The idea is to start from a set of pairs of parallel monomorphisms to N^f. An adjacency criterion selects which of these pairs should become (co)equalised by c. When properly applied, this idea yields a final reduction that is itself couniversal, in particular, it is unique. It is computed by a sequence of couniversal constructions, namely coequalisers. This computation is very efficient because it requires neither searching nor backtracking and allows important optimisations.

This construction allows variations in three dimensions:

- the morphisms of the category determine the similarity criterion of the subnets which might become merged
- the adjacency criterion
- a choice between pairs yielding conflicting coequalisers

The last variation choice may lead to the loss of uniqueness. These variations allow flexible adaptation of the construction to a specific reverse-engineering task. A prototypical implementation and practical applications are described in Chapter 4.

3.5.1 Coloured Nets

Definition 30. C*NET is one of the three following comma categories of unitary foldings:

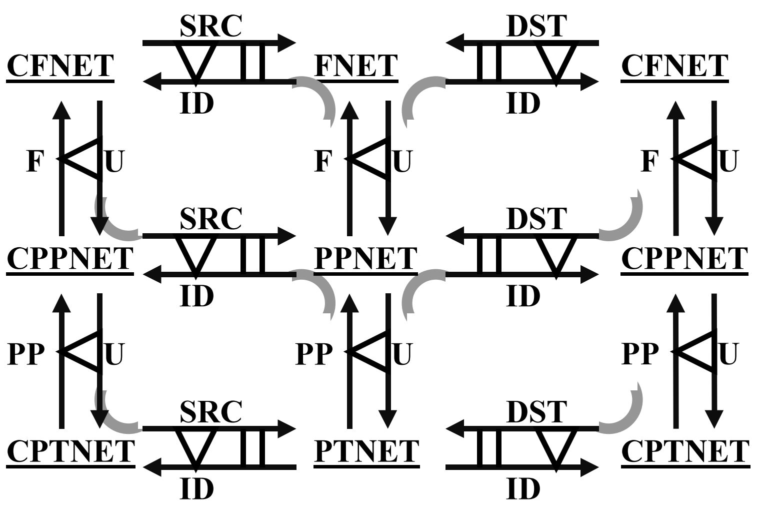
- CPTNET in PTNET
- CPPNET in PPNET and
- CFNET in FNET.

The definition can be expanded by saying that a coloured net of C*NET is a unitary folding C: $N^s \to N^d$ from a source net src C = N^s to a destination net dst C = N^d. A morphism f: C→C' is a pair of morphisms f^s: src C→src C' and f^d: dst C→dst C' which together with C and C' form a commutative diagram in PTNET, PPNET or FNET respectively.

$$\begin{tikzcd}
C \arrow[r, "f"] & C' \\
\text{src } C \arrow[r, "f^s"] \arrow[d, "C"'] & \text{src } C' \arrow[d, "C'"] \\
\text{dst } C \arrow[r, "f^d"] & \text{dst } C'
\end{tikzcd}$$

Figure 10. A colour morphism.

Proposition 31. src and dst extend to functors SRC and DST: C*NET→*NET. DST is left adjoint to ID (with ID N = Id_N), forming a reflection, and SRC is right adjoint to ID, forming a coreflection. The functors PP and F and their adjunctions with U lift to the categories C*NET and commute with the former adjunctions. ⇨118◈



The proposition is shown alongside with grey arcs depicting the compositionality of adjunctions.

Proposition 32. CPPNET and CFNET both are cocomplete and complete. ⇨119◈

Although this definition sounds quite different from the usual definition of coloured nets, such as that of CPN [Jen92], in reality they are not that far apart.

Before a formal discussion look at the example of a CPN in Figure 11. There are three places all with the same colour set $\mathbb{Z}_5$ the field of the integers modulo 5. The arcs carry expressions whose variables are bound around a transition. A transition occurrence takes from the left places a number x and a number y and deposits the quotient in the place at the right side. Additionally the transition has a guard – it is only allowed to occur if y ≠ 0.

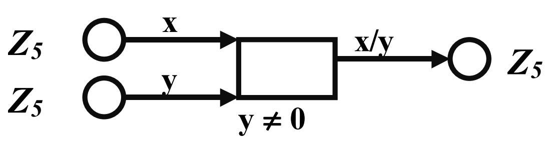


Figure 11. A coloured net.

A given CPN N^J translates to a

C*NET C: $C^s \rightarrow C^d$ as follows. Places of N^J become places of the destination net C^d, the places of N^s are the token elements (pairs of a place and a compatible colour) with $C_\beta(p, c) = p$. Similarly, the binding elements (pairs of a transition and a transition mode permitted by the guard expression) become the transitions of N^s. The pre and post-functions of the source net are derived from the arc-expression function E^J (arc identities are ignored to simplify notation) as follows:

$pre^s((t,b), (p,c)) = E^J(p, t)\ (b)\ (c)$

$post^s((t,b), (p,c)) = E^J(t, p)\ (b)\ (c)$.

Finally

$T^d = \{(t, C\ pre^s\ (t, b), C\ post^s\ (t, b)) \mid \text{for each binding element } (t, b)\}$

$pre^d(t, u, v) = u$

$post^d(t, u, v) = v$

$C(t, b) = (t, C\ pre^s\ (t, b), C\ post^s\ (t, b))$ for each binding element (t, b)

complete the definition of N^d and clearly turn C into a unitary folding.

If t uniquely determines $(t, C\ pre^s\ (t, b), C\ post^s\ (t, b))$, this yields a perfect correspondence: the net of N^J is C^d and C is the unfolding of the coloured net to the equivalent flat net, which is C^s. In general, however, the CPN transitions must be split up to force C to commute with pre and post. This splitting is necessary because C*NET enforces a coherence condition (by the definition of a folding) whereas CPN delegates such a coherence condition to an unspecified typing system.

In conclusion, CPN and C*NET differ in the following ways:

- C*NET abstracts away the typing system in CPN. Typing is indispensable for programming but is neglected here as a concept which is orthogonal to Petri net theory.
- For lack of space, we defined only coloured nets and not systems which include an initial marking as CPN does.
- C*NET enforces coherence between the coloured net and its unfolding whereas CPN delegates this coherence to an unspecified typing system.
- CPN lacks the categorical machinery which is a strength of C*NET.

3.5.2 Hierarchical Nets

Hierarchical nets support additional structuring methods. Beside colouring they allow (copies of) components to be composed, and hence permit layered design and analysis. A general and simple way to attain this within the current categorical framework will now be shown:

Definition 33. The categories HPTNET, HPPNET and HFNET respectively consist of

- finite commutative diagrams as objects, and
- natural transformations as morphisms

in PTNET, PPNET and FNET respectively.

Natural transformation means that a correspondence is defined between the objects of the two diagrams, and corresponding objects are connected by morphisms to make the combined diagram commutative. This is a generalisation of coloured nets that allow only diagrams with a single arrow. The purpose is to use colourings where appropriate and forget about them elsewhere. The diagram would be annotated as follows (for instance):

- this arrow is a colouring of database entities
- this is a pushout square composing two components

- this is a simulation arrow defining concurrent behaviour
- this arrow is a place invariant ensuring that no clients get lost

To model an existing hierarchical net class, only a few types of arrows are needed. For instance HCPN [Je92] would need monomorphisms for subnet identification and unitary foldings for place fusions and colourings. Place-transition morphisms may be used for clustering. This definition of hierarchical nets is obviously very general and a specific tool would have to select a subcategory. However, this is beyond the scope of this paper.

Within this framework, the task of reverse engineering can be expressed thus: given a net N^f , find a hierarchical net that specifies the design of N^f. The hierarchical net is a diagram consisting of colouring foldings, clustering morphisms, monomorphisms identifying subsystems etc.. The task thus reduces to the following subtasks: to compute colourings, to find clusterings, to identify subsystems etc. and to select and compose a meaningful diagram from them.

Here, we concentrate on the first subtask. Chapter 4 will show that folding-based methods yield powerful new analysis methods for conventional reverse engineering. For the remaining subtasks, on the other hand, many clustering-based methods from traditional engineering apply. They usually use clustering methods, because these are the fundamental method to compose complex systems from subsystems.

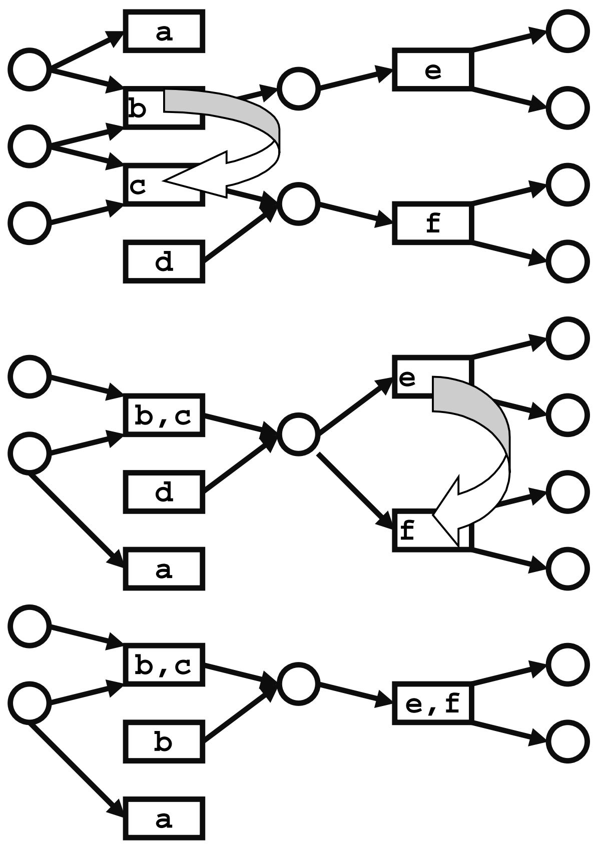


Figure 12. Reduction by folding.

3.5.3 Iterated Couniversal Constructions

For a given net N^f, we must compute a colouring morphism c: $N^f \rightarrow N^u$. The basic idea may be illustrated by another use of the word folding. Just like folding a shirt, two adjacent parts which fit together are overlaid. Figure 12 shows this process for Petri nets. The reduction works iteratively, i.e. it may overlay fitting parts when they become adjacent, and it stops when there are no more adjacent fitting parts.

Only similar transitions may be overlaid, whereas there is no such restriction for places. The process therefore relies on the bipartite structure of the graph. This folding process is formalised as follows:

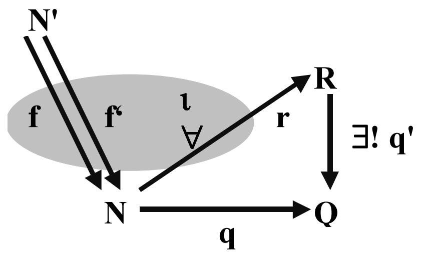


Figure 13. A maximal ι reduction R.

Definition 34. Let ι be a relation on morphisms with
if (f, f')∈ ι then (gf, gf')∈ ι for each compositional morphism g.

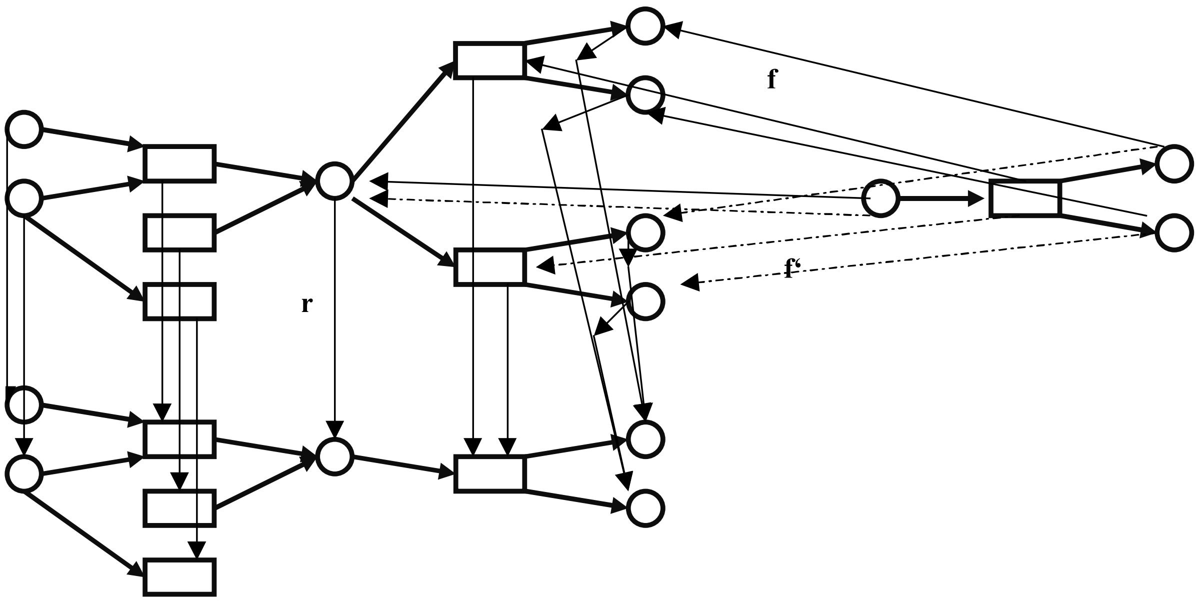


Figure 14. The lower reduction from Figure 12 as coequaliser.

A morphism r: N→R is an ι reduction of N iff

$\forall$f, f': N'→N: if (r f, r f')∈ι then r f = r f'

The maximal ι reduction of N is an ι reduction which is couniversal for all ι reductions.

The definition is visualised in Figure 13, which is commutative except for the parallel arrows f and f'. The idea is that two morphisms which are related by ι have the same source and destination, and their images should become overlaid. Figure 14 shows that this has the same result as the folding reduction from Figure 12.

Proposition 35. A cocomplete category possesses all maximal ι reductions. ⇨120◈

For finite nets of PPNET or FNET, the proof contains an algorithm: just compute one coequaliser after the other. When no further reduction is possible, the maximal ι reduction is reached. The next proposition is crucial for efficiency.

$$\begin{tikzcd}
 & N_h \arrow[d, shift right, "h"'] \arrow[d, shift left, "h'"] & \\
N_f \arrow[ur, "f_h"] \arrow[dr, shift left, "f'"] \arrow[dr, shift right, "f"'] & & N_g \arrow[ul, "g_h"'] \arrow[dl, shift right, "g"'] \arrow[dl, shift left, "g'"] \\
 & N & \arrow[r, "e"'] & N_e
\end{tikzcd}$$

Figure 15. The Factorisation.

Proposition 36. If ι is a relation on morphisms as in Definition 34 and κ is the least relation fulfilling points (i) to (iii) from below then an r is an ι reduction of N iff it is a κ reduction.

(i) ι⊆κ

(ii) if (f, f')∈κ then (f g, f' g)∈κ for each compositional morphism g

(iii) if $\forall$ (f, f')∈κ, (e g, e g')∈κ and h, h', f_h and g_h are morphisms fulfilling the points (a) to (d) from below then (h, h')∈κ (refer to Figure 15):

(a) f = h f_h and f' = h' f_h,

(b) g = h g_h and g' = h' g_h,

(c) e is the coequaliser of f and f'

(d) $\forall$ x, x': N_h →N_x holds x = y iff (x f_h = y f_h and x g_h = y f_h) ⇨120◈

To reduce a relation ι the proposition is applied in reverse: pairs generated by right multiplication (ii) or union (iii) are superfluous. For many applications it is sufficient to use morphisms from nets consisting of the environment of a single transition. This yields an efficient computation because it allows to work locally.

3.5.4 Variations

These universal constructions may be modified in three dimensions:

- Adjacency: The relation ι decides about the necessary adjacency criterion for subnets - including ignoring adjacency completely.
- Similarity: The morphisms of the category determine which subnets are merger candidates. Should subnet isomorphism be weakened or strengthened by additional requirements?
- Choice: which subset of the mergers allowed by the above two criteria is selected? This leads to similar constructions that may not be universal or unique.

There are several ways of defining adjacency:

Definition 37. Define relations on parallel morphisms f, f': $N' \rightarrow N$ by:

- **overlap**: $\text{im } f_\beta \cap \text{im } f'_\beta$ is not empty
- **fixpoint**: $\exists x' \in X_{N'}$ with $f\ x' = f'\ x'$
- **common intersection**: $\{x' \in X_{N'} \mid f\ x' = f'\ x'\} = \text{im } f_\beta \cap \text{im } f_\beta' \neq \{\}$
- **clean intersection**: (if $f_\beta\ x' \in \text{im } f_\beta'$ or $f_\beta'\ x' \in \text{im } f_\beta$ then $f\ x' = f'\ x'$) and $\text{im } f_\beta \cap \text{im } f_\beta' \neq \{\}$.

These four relations are totally ordered with overlap the coarsest and clean intersection the finest relation. As Figure 16 shows, the first two definitions may still unnecessarily disturb the 'isomorphic transition' picture. We were unable to find any such examples of clean intersection which coincides with common intersection for monomorphisms. However, the two last relations are not preserved by left multiplication, as is required by Definition 34.

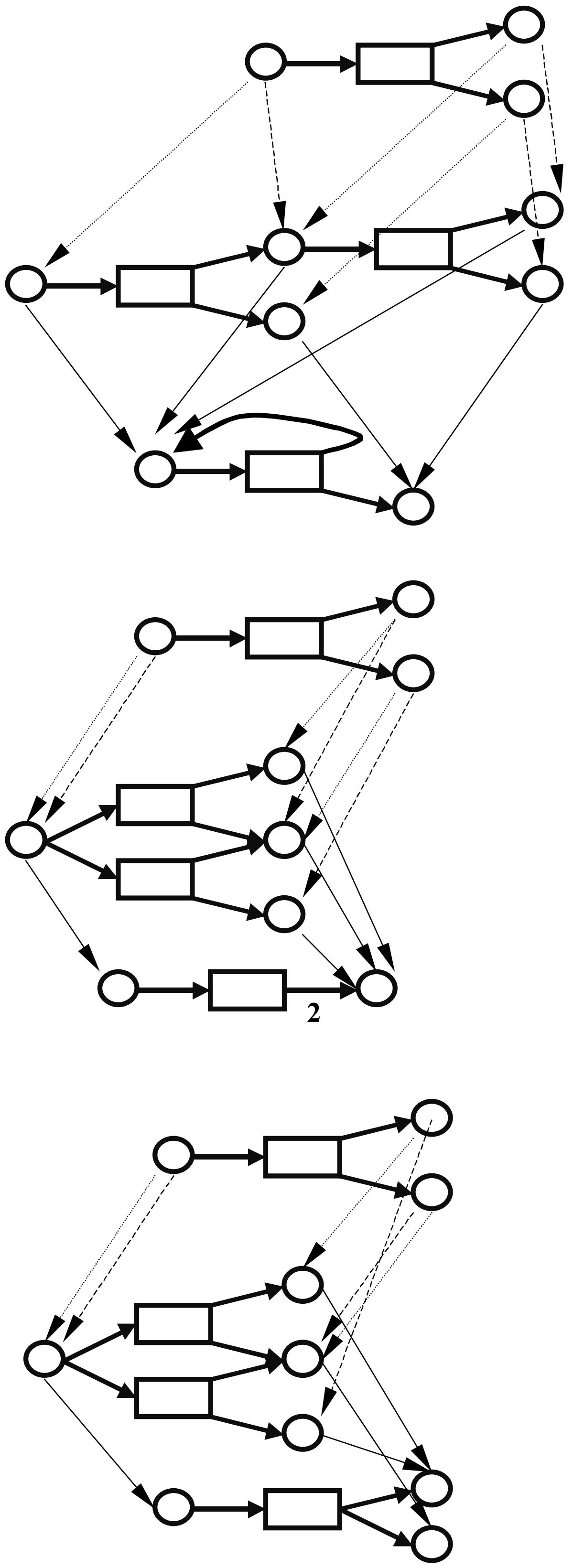


Figure 16. Bad examples for overlap (top) and fixpoint (middle) not occurring with clean intersection (bottom).

They may be used for more general reductions.

These adjacency definitions only ensure that a single reduction step is locally clean. We will need additional methods to get clean reductions for the whole net. For this Definition 34 and the following propositions are formulated purely categorically which allows selecting a subcategory of PPNET or FNET. Chapter 4 will use a preliminary analyses to select a subset of morphisms and then will compute the reduction in the resulting subcategory.

Up to now, a reduction has equalised all morphisms related by ι. This is in line with universal constructions, avoids combinatorial explosion and yields a unique solution. Sometimes, however, it is crucial to select a pure variant and avoid mixtures. An algorithm using arbitrary choice would be more appropriate in this case. It would, for instance, not lead to over-simplification as in Figure 17. We formalise these ideas as follows:

Definition 38. Let ι be an arbitrary relation on morphisms. A morphism r: N→R is an ι choice reduction of N iff

for each f, f': N'→N with (rf, rf')∈ι there exist morphisms g, g': N"→N with

∀ x, y: (x f = y f iff x g = y g) and (x f' = y f' iff x g' = y g'),

(r g, r g')∈ι and r g = r g'

With this definition universal properties and uniqueness are lost even for maximal (with regard to connecting morphisms) choice reductions. The optimisations offered by Proposition 36 may remain valid, at least as heuristics. But the same algorithm that computes coequalisers iteratively still works.

There are many interesting questions about choice reductions. What about the limit of all (maximal) choice reductions of a net? What are the common properties of all reductions? In section 4.6.1 we propose that all such reductions be computed by relational algebra in a product representation which would allow reverse-engineering information to be extracted from the set of all reductions.

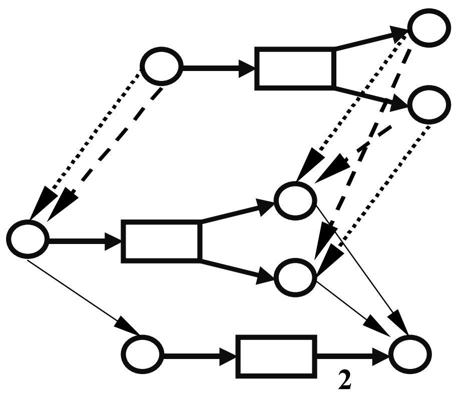


Figure 17. A single choice would be better.

3.6 From Structure to Behaviour and Semantics

We have built a bridge from clustering to folding, but a final point is still missing: do we obtain the strong relationships to semantics which are typical for folding-based Petri-net categories? A positive answer will affirm that we have built a folding category in a deep sense and not only superficially. Semantics is the final purpose of all work with Petri nets: nets and systems represent a compact formulation of possible behaviours or semantics. Strong relationships to semantics allow semantic analyses to be reduced to simpler structures, which is often crucial for avoiding combinatorial explosions.

A Petri net is a static structure, no more than a bipartite graph. A Petri system is a net plus an initial marking which enables a token game representing the behaviour of the system. The main interest in net structures is to understand the behaviour of nets. Hence tight relationships between nets and systems are crucial: They allow the properties of a system to be derived from a static net. In particular, we would like to find the bridge from clustering to folding for systems.

The dialectic between structure and behaviour is a basic principle of Petri-net theory, so it would be useful to formulate it categorically. To the author's knowledge, however, there has so far been no reference to this being done in the literature.

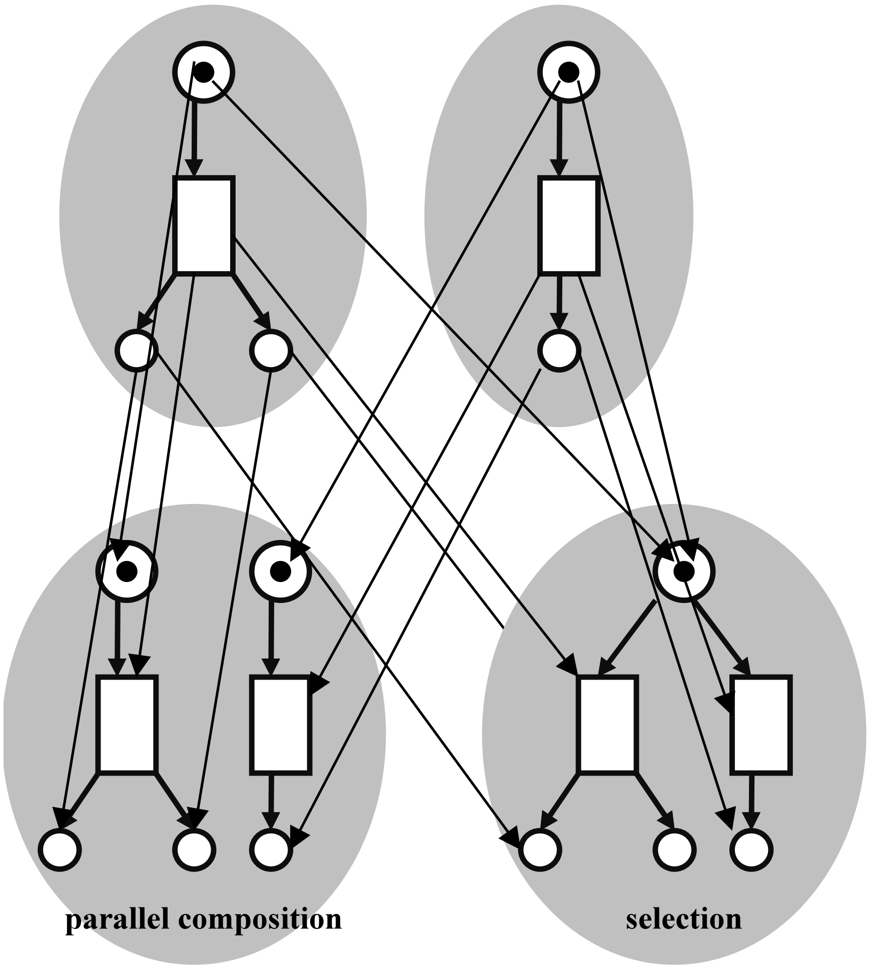


Figure 18. Parallel composition and selection.

For each net category, a corresponding system category PTSYS, PPSYS or FSYS respectively is defined. The objects are always the same. As a rule, a system is a net with an initial marking. Morphisms are novel. They stem from the corresponding net category PTNET, PPNET or FNET respectively, with the restriction that the image of the initial marking must be smaller than or equal to the initial marking of the destination system. This represents a generalisation of the equals relationship usually applied. This is not a big change for the simulation: some tokens may simply remain immobile. However, morphisms can now express parallel composition and not only selection, as is shown in Figure 18.

This definition yields a coreflection between nets and systems. More precisely, there is a coreflection between each net category and the corresponding system category. Furthermore, the adjunctions from PTNET to PPNET and from PPNET to FNET lift to systems. Together, these seven adjunctions form a commutative diagram of adjunctions. Very powerful relationships therefore exist between the system and net categories, and the bridge from clustering to folding extends naturally from nets to systems. This web of adjunctions expresses the Petri-net dichotomy of structure and behaviour. A (co)universal construction of systems reduces to a (co)universal construction of the initial marking plus a construction in the net category.

A frequently used semantics for Petri nets is the reachability graph. The unfolding of a system to its step-reachability graph turns out to be a functor SM: PPSYS→SM with SM being the category of state machines, a subcategory of PPSYS. This immediately shows how morphisms simulate step sequences and how they transfer behavioural properties such as liveness or boundedness.

Reachability semantics simply reflect the possible moves of a system whereas occurrence semantics encompass causality and branching. The literature describes a co-reflection between folding-based system categories and occurrence nets - more precisely, there are coreflections to different subcategories of such a system category but none to the whole category. This same phenomenon is found for FSYS.

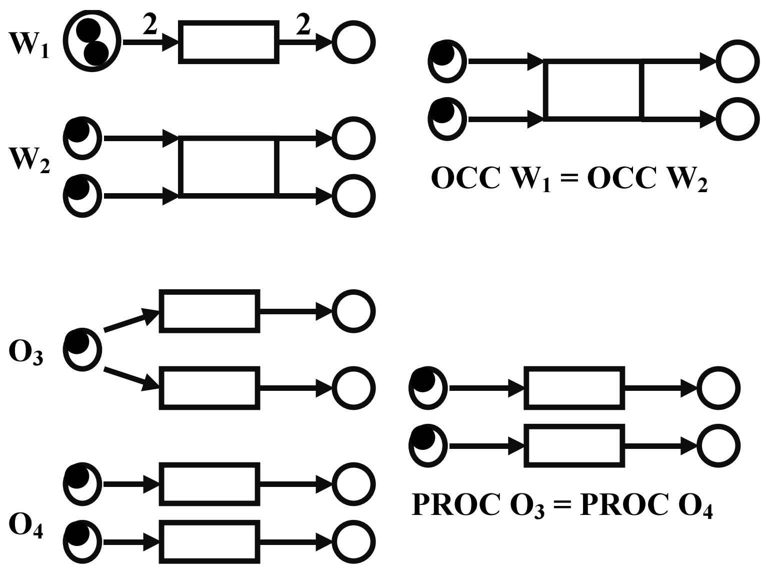


Figure 19. A weighted occurrence system W_1, a safe occurrence system O_3, processes W_2 and O_4 and their unfoldings.

Moreover, we propose an alternative approach. Instead of restricting the system category, we generalise occurrence systems. This yields a similar coreflection between FSYS and WOCC, the category of weighted occurrence systems. WOCC does not distinguish tokens with the same history as safe occurrence systems do. This means that tokens are only differentiated if they have different causal dependencies. Hence in the range of individual and collective token philosophies, ordinary occurrence systems individualise all tokens whereas WOCC selects the point between the two poles which reflects causality and branching. Figure 19 shows examples for the different systems and unfoldings.

The adjunctions from PTSYS to PPSYS, from PPSYS to FSYS and from FSYS to WOCC compose. Thus weighted occurrence systems not only explain the behaviour of folding systems but also of place-preserving and place-transition systems.

This work does not need Petri-net semantics for reverse engineering – indeed, we state the thesis that bipartite weighted graphs are already providing a powerful modelling metaphor for software engineering. Our goal in this last section of the net-theoretical part of this dissertation is to demonstrate that the defined categories are folding-based in a deep sense, i.e. they have strong categorical connections to behaviour and semantics. To show this, it is sufficient to give some highlights. Completeness is not aimed for in any way. Rather we hope to encourage further research because we assume that there are many more features to discover.

3.6.1 Net Systems

Definition 39. The categories PTSYS, PPSYS and FSYS respectively consist of

- objects S = (N, I) with N a net (of PTNET) and $I \in 1S\ P_N$ the initial marking
- a morphism f: S→S' is a morphism N→N' of PTNET, PPNET and FNET respectively fulfilling $f\,I \le I'$.

*SYS and *NET symbolise any of these pairs of corresponding categories.

The initial marking must map to a marking contained in the initial marking of the destination net. This is a slight generalisation over the usual equal. The generalisation adds modelling flexibility, allows zero-objects and adjunctions between nets and systems. Hence a simpler and more powerful theory is obtained.

	[GLT79]	PTSYS	PPSYS	FSYS	[Win84a]
$\forall p \in P$: f p	$\in X'$	$\in \mathbb{N} X'$	$\in \mathbb{N} P'$	$\in \mathbb{N} P'$	$\in$ MS P'
$\forall t \in T$: f t	$\in X'$	$\in \mathbb{N} X'$	$\in \mathbb{N} X'$	$\in \mathbb{N} T'$	$\in T' \cup \{\text{undefined}\}$
f I	-	$\le I'$	$\le I'$	$\le I'$	$= I'$

Figure 20. Comparison of a morphism f: S→S' in different categories.

The objects in the three categories *SYS and in [Win84a] are the same up to technical restrictions such as finiteness or emptiness. [GLT79] provides no initial markings and no arc weights. For morphisms the situation is tabulated in Figure 20 (compare also Figure 8).

Definition 40. Let S = (N_S, I_S) and S' be systems, f: S→S' be a morphism and define the following functors:

- NET: *SYS→*NET maps S to N_S and f to f.
- IP: *SYS→*NET maps S to the net (PTNET supp I_S) and f to the restriction of f to IP S.
- IM: *SYS→*SYS maps S to (IP S, I_S) and f to IP f.
- SYS0: *NET→*SYS maps a net N to (N, **0**) and a morphism to itself.

NET forgets the initial marking, IP retains only the places marked by the initial marking, IM retains only the initial marking and SYS0 adds a zero initial marking.

Proposition 41. SYS0: *NET→*SYS forms a coreflection with NET. PP and F lift to systems yielding the commutative adjunction diagram of Figure 21. ⇨121◈

As announced in the introduction this web of 7 adjunctions expresses the Petri net dichotomy of structure and behaviour combined with the bridge from clustering to folding.

$$\begin{tikzcd}
\underline{\textbf{FNET}} \arrow[r, shift left, "\textbf{SYS0}"] \arrow[d, shift left, "\textbf{U}"] & \underline{\textbf{FSYS}} \arrow[l, shift left, "\textbf{NET}"] \arrow[d, shift left, "\textbf{U}"] \\
\underline{\textbf{PPNET}} \arrow[u, shift left, "\textbf{F}"] \arrow[r, shift left, "\textbf{SYS0}"] \arrow[d, shift left, "\textbf{U}"] & \underline{\textbf{PPSYS}} \arrow[u, shift left, "\textbf{F}"] \arrow[l, shift left, "\textbf{NET}"] \arrow[d, shift left, "\textbf{U}"] \\
\underline{\textbf{PTNET}} \arrow[u, shift left, "\textbf{PP}"] \arrow[r, shift left, "\textbf{SYS0}"] & \underline{\textbf{PTSYS}} \arrow[u, shift left, "\textbf{PP}"] \arrow[l, shift left, "\textbf{NET}"]
\end{tikzcd}$$

Figure 21. Proposition 41.

Proposition 42. IM: *SYS→*SYS has a right adjoint, namely MM2I for PTSYS and PPSYS and FMM2I for FSYS. ➩121◈

$$\begin{tikzcd} {*SYS} \arrow[r, shift left=2, "IM"] \arrow[r, phantom, "\nabla"] & {*SYS} \arrow[l, shift left=2, "{*MM2I}"] \end{tikzcd}$$

The functor *MM2I crumples a net, saving only the initial marking and inventing transitions to allow morphisms. The net of MM2I S is infinite if the initial marking contains at least one place.

Lemma 43. There is a natural transformation: ι: IM→ID_{*SYS} with all ι unitary monomorphisms.

Proof: Define $ι_S$ as the injections of the places of the initial marking of S into N_S. Clearly, ι preserves the initial marking and ι is natural ♦

Figure 18 shows that system morphisms allow to model parallel composition which is the same as disjoint union. The latter is the coproduct for nets whereas the two systems do not have a coproduct. Systems are neither cocomplete nor complete as the underlying nets are.

Proposition 44. A diagram D in *SYS
- has a colimit iff IM D has a colimit j: IP D→J and the combined diagram NET (j: IM D→J, ι: IM D→D) has a colimit in *NET
- has a limit iff IM D and NET D have a limit

As prerequisite the universal construction for IM D only needs naturality, uniqueness may be dropped. ➩122◈

The quintessence of this proposition can be stated as: A universal construction in *SYS is reduced to the universal construction of the initial marking and a universal construction in *NET. A diagram in FSYS or PPSYS hence has a colimit or a limit iff the initial marking allows it.

3.6.2 Reachability and Liveness

In the last section we introduced system categories that reflect the behavioural aspects of Petri nets. As announced we turn now to the study of semantics.

The semantics of reachability graphs are frequently used. The nodes of the graph are the states – e.g. markings – of the system. The arcs reflect state changes. This is a very intuitive abstraction of a system as a set of states and allowed state changes. The transfer of the initial marking by system morphisms ensures that if the source net can do a move the destination net can do the corresponding move. Reachability graphs represent the possible moves of a system. A place-transition morphism simulates the occurrence of a transition in the source net by
- an occurrence of a transition in the destination net
- ignoring it, if it is mapped to the inside of a node
- starting a transition if it is mapped to the pre side of a destination transition
- finishing a transition if it is mapped to the post side

This is quite a complicated simulation – violating in fact a basic principle of Petri net behaviour, namely, the atomicity of transition occurrence. This was one of the reasons to introduce nets with simpler morphisms.

In order to define a functor from systems to reachability graphs a destination category is needed. There is a natural choice: reachability graphs can be represented as special Petri nets:

Definition 45. <u>SM</u> the category of state machines is the full subcategory of <u>PPSYS</u> with the objects fulfilling

- all arc weights are one and each transition has exactly one input and one output place
- $I = 1\ p_I$: the initial marking consists of a single token on a place p_I
- $X = p_I\ F^*$ with F^* the transitive closure of the flow-relation F: neither dead transition nor never-marked places.

There are two common ways to unfold a system to a state machine:

- sequential: a transition in the state machine corresponds to the occurrence of a single transition
- step: a transition in the state machine corresponds to the occurrence of a transition step

In our framework it is easier to handle the second case:

Proposition 46. Unfolding the steps of a system to a state machine extends to a functor SM: <u>PPSYS</u>$\rightarrow$<u>SM</u>. ⇨124◈

This functor allows to transfer properties such as liveness or boundedness between two systems. Again we only give some samples:

Proposition 47. Let f: $S \rightarrow S'$ be a place-preserving morphism and t a transition of S with $f_\beta\ t \in T'$ and m and m' reachable markings of S then

- if m' is reachable from m then also fm' from fm
- if f_β t (even f t suffices) is dead at fm then also t at m
- if S is live and f epimorphic then also S' is live
- if S' is bounded at f_β p by β' then S is bounded at p by β / (f_γ p)

The proof uses f to map step sequences ♦

Proposition 48. SM: <u>PPSYS</u>$\rightarrow$<u>SM</u> has neither a left nor a right adjoint. ⇨125◈

This proposition shows that SM has its limitations. Other morphism definitions get adjunctions here but our definition enforces a tight relationship with the underlying graph. The missing adjoint for SM is a disadvantage of this design decision. Reachability semantics abstracts too much from the graph of the net system to allow an adjunction. But for handling liveness there are alternatives to the functor SM:

Proposition 49. Let <u>*ND</u> the full subcategory of <u>*SYS</u> with neither dead transitions nor never marked places. The functor ND: <u>*SYS</u>$\rightarrow$<u>*ND</u> removing dead transitions and never marked places forms a coreflection with the underlying functor. ⇨125◈

$$\begin{tikzcd}
\underline{\text{*ND}} \arrow[r, shift left=2, "U"] \arrow[r, phantom, "\Vert\ \vee"] & \underline{\text{*SYS}} \arrow[l, shift left=2, "ND"]
\end{tikzcd}$$

In conclusion, the developed framework integrates with reachability semantics and basic Petri-net properties such as liveness and boundedness.

3.6.3 Processes

Process semantics describe the possible moves of a system as reachability semantics does. But additionally, behaviour is explained in terms of causality and branching. Hence process semantics differentiate between two occurrences of a transition if they consume tokens produced by different transition occurrences. Put otherwise, a transition occurrence is tagged with the origins of the consumed tokens: with its history.

Figure 22 shows two nets with the same unfolding in state machines but different processes. History and causality are clearly shown in the process pictures. The interrupted lines in the processes show maximal place cuts which correspond to reachable markings. Processes fit

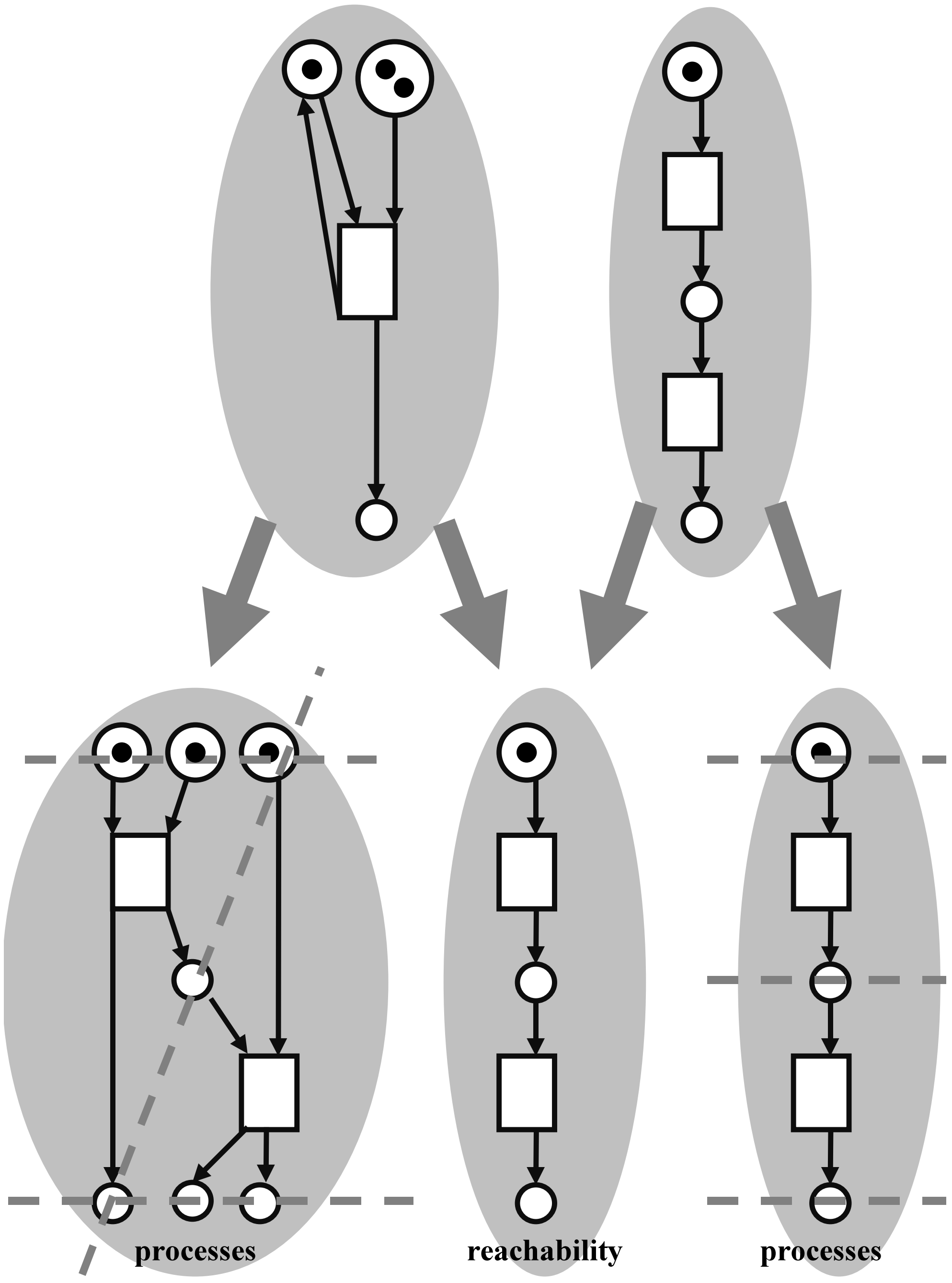


Figure 22. Process and reachability semantics.

smoothly in our framework because they are special morphisms:

Definition 50. A system is safe iff all arc weights are one and any reachable marking is a set (rather than a multiset).

Definition 51. The category WOCC of weighted occurrence systems is the full subcategory of FSYS whose objects O fulfil

- $\forall p \in P_O$: $| {}^\bullet p | \leq 1$
- supp $I_O = \{p \in P_O \mid | {}^\bullet p | = 0\}$
- the flow relation F is acyclic and $X_O = (\text{supp } I_O)\, F^*$
- $\forall x \in X_O$: F^* x is finite
- R has no dead transitions

The category OCC of (safe) occurrence systems is the full subcategory of WOCC of safe systems and the category PROC of processes is the full subcategory of OCC with

- $| p^\bullet | \leq 1$ for all places p

A unitary folding o: $O \rightarrow S$ is called a (weighted, safe) occurrence system or a process respectively of the system S.

Processes are acyclic safe nets, thus the initial marking is a set instead of a real multiset. Each place not in I has a unique input transition able to deliver a single token on it and a unique output transition that may consume this token later.

Each node of a weighted occurrence system is assigned a depth:

- depth 0 for places in I and depth 1 for transitions with empty preset.
- depth i+1 for a transition with the maximal depth of an input place equal i
- depth i for a place produced by a transition of depth i

The finiteness condition guarantees that each node has finite depth.

Definition 52. A maximal place cut of X is a maximal subset of P such that no two elements are related by F^*.

Maximal place cuts correspond to reachable marking of the process or the system:

Lemma 53. The image of a finite maximal place cut of a process is a reachable marking of the system. Any step sequence of a system is the image of a step sequence in an appropriate process. ⇨125◈

3.6.4 Weighted Occurrence Systems

The set of all processes is a bit unhandy. It would be nicer to fold a set of processes into a single net. This is exactly what weighted occurrence systems are for. First we need a generalisation of a maximal place cut:

Definition 54. A cut step is a multiset τ of transitions with pre $\tau \leq I_0 + \text{post } \tau$ and $I_0 + \text{post } \tau - \text{pre } \tau$ is a cut.

The definition is motivated by the following lemma. It also shows that cuts as defined above coincide with those defined for processes:

Lemma 55. The image of the transitions of a process of a weighted occurrence system is a cut step and each cut step is such an image. Hence a marking is reachable iff it equals $(I - \text{pre}\,\gamma + \text{post}\,\gamma)$ for a cut step γ. A process of a weighted occurrence system of a system S yields a process of S. ⇨126◈

This lemma gives a simple computation of the processes of a weighted occurrence system: expand the cut steps into processes. As figured out in the proof, expansion means to multiply each transition t $\sigma(t)$ times and each place p $(I + \text{post}\ \sigma)(p)$ times and to select compatible connections. This also works for a system in general if an appropriate occurrence system is available. This is accomplished by:

Proposition 56. There is a functor WOCC: <u>FSYS</u>→<u>WOCC</u> which is right adjoint to the underlying functor U forming a coreflection.. ⇨127◈

$$\begin{tikzcd}\underline{\text{FSYS}} \arrow[r, bend left, "\text{WOCC}"] \arrow[r, phantom, "\Lambda\ \sqcap"] & \underline{\text{WOCC}} \arrow[l, bend left, "U"]\end{tikzcd}$$

Lemma 57. The processes of a system S are in bijection with the processes of WOCC S.

Proof: A unitary folding r onto a cut step of WOCC S gives a process ε_S r by Lemma 55. Because a process is a weighted occurrence system it factorises through WOCC S. This relationship is bijective because ε_S is a universal arrow from U to S which equivalent with adjointness ♦

Hence the processes of S may be computed as the processes of the weighted occurrence system WOCC S. Also, Lemma 55 yields an easy characterisation of the processes of a system. WOCC S is a representation of all the processes of a system S in a single net.

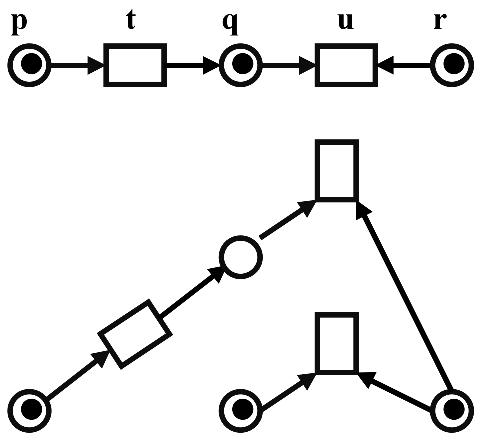


Figure 23. A system (top) and its unfolding (bottom).

Figure 23 illustrates the unfolding by WOCC at an example taken from [GP95]. An occurrence of transition t creates a second token on place q. According to the individual token philosophy there are two possibilities for the occurrence of u:

- either the initial token on q is consumed: such an occurrence is concurrent with t
- or the token produced by t is consumed: such an occurrence is causal dependent from t.

This is reflected by the unfolding in Figure 23: there is one transition for each of these two possibilities. Furthermore, this unfolding coincides with the unfolding into safe occurrence nets (as defined e.g. in [MMS92]). The two unfoldings coincide for safe and for semi-weighted nets (systems with the weight of any output-arc of any transition equal 1 and the initial marking a set [NS98]).

But they differ on any system that is not semi-weighted. This is shown in Figure 24. It uses the same net as the previous example, but, there are two initial tokens on place r. The WOCC unfolding yields the same net as before, only the initial marking is adapted. However, the safe unfolding puts the additional initial token in a additional place and consequently needs to differentiate between further types of occurrences of transition u. This means that

- the unfolding into safe occurrence systems radically individualises tokens, they have a unique identity
- the WOCC-unfolding differentiates only tokens that are located in different places or have a different causal history
- in the collective token philosophy tokens on the same place are indistinguishable.

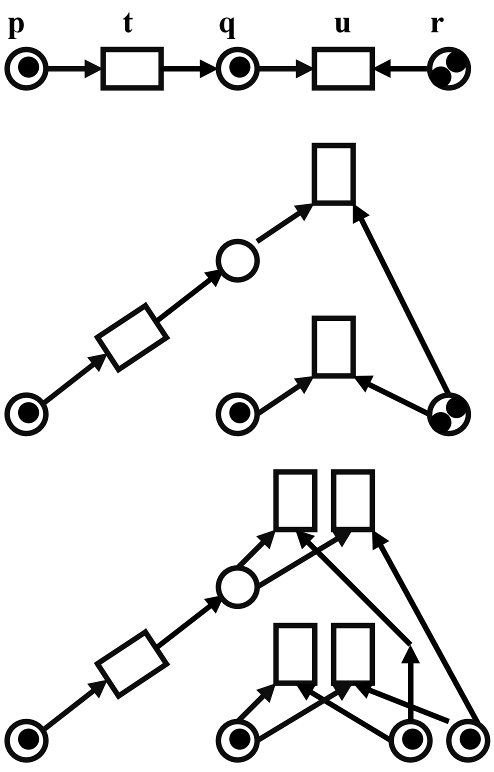


Figure 24. A system (top), its WOCC-unfolding (middle) and its safe-unfolding (bottom).

The radical individualisation of tokens by safe unfoldings introduces inconveniences. First, the unfolding yields conflicts between tokens on the same place with the same history which seems strange in the case of anonymous tokens. Secondly, section 3.6.5 will argue that there might be no 'reasonable' adjunction between systems and safe occurrence systems. Thirdly, using collective token philosophy in a pure form [Bru98] builds attractive categorical connections between behavioural, algebraic and logical models – whereas the corresponding connections in the individual token philosophy seem to be less simple, maybe because they a radical instead of a 'natural' individual token philosophy. Finally, safe unfoldings do not easily cope with variable initial markings. Such markings often occur when a net is used as a function transforming tokens from 'input-places' to tokens on 'output-places' (e.g. [Feh93], [Ess97]).

This is handled elegantly by our WOCC-unfolding. Figure 25 shows how to convert p and r in our example net to input-places. Here, p and r may receive an arbitrary number of tokens from two initial transitions with empty presets. Such transitions are disallowed for safe unfoldings. But they are no problem for WOCC.

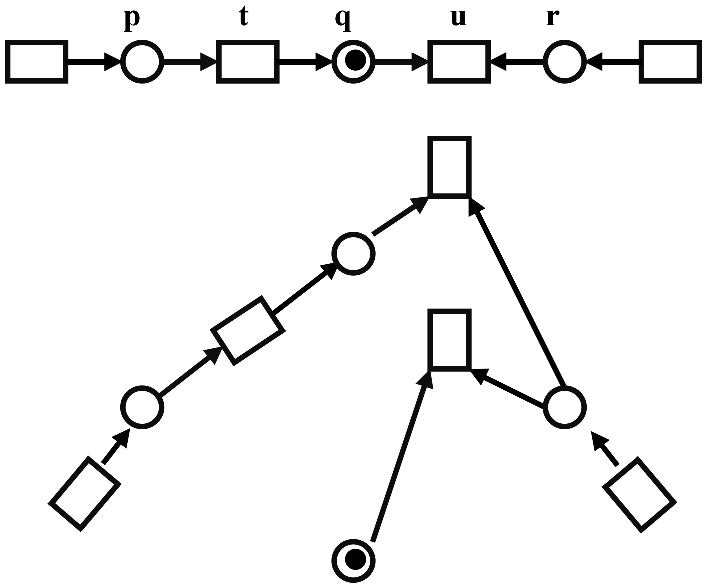


Figure 25. A system (top) with variable initial marking on input-places p and r and its WOCC-unfolding.

Safe occurrence systems realise a radical individual token philosophy. However, if the main interest is in causality and branching it is reasonable to look for the position between the two poles of individual and collective token philosophy that best serves this interest. It is exactly this intermediate position that weighted occurrence systems realise: they individualises tokens iff they have a different causal history, not generally as safe unfoldings do. In this sense, weighted occurrence systems represent the pure semantics of causality and branching for anonymous tokens.

3.6.5 Safe Occurrence Systems

The category of weighted occurrence system is new. Known from the literature are occurrence nets. As a bridge to them we need decorated occurrence systems:

Definition 58. A place decoration Φ of a safe system S is a function Φ: $P_S \rightarrow \mathbb{N}^+$ with

$\forall t \in T_S$: $\Phi(t^\bullet) = [1, | t^\bullet |]$

$\forall p \in \text{supp } I_S$: $\Phi(p) = 1$

Thus, a place decoration numbers the output places of each transition of a system.

Definition 59. D = (pp_D, I_D, Φ_D) is a decorated occurrence system if (pp_D, I_D) is a safe occurrence system and Φ_D a decoration of the places of S. P_D. <u>DEC</u> is the category of decorated occurrence systems. A morphism f: D→D' of <u>DEC</u> is a morphism of the underlying save occurrence systems which fulfils:

$\forall p, q \in P_D \cap \text{def } f_\beta$ with ${}^\bullet p = {}^\bullet q \neq \{\}$: ($\Phi_D\, p < \Phi_D\, q$ iff $\Phi_D'\, f p < \Phi_D'\, f q$)

Definition 60. <u>FSYS1</u> is the full subcategory of <u>FSYS1</u> of systems with the initial marking being a set. Similarly, <u>WOCC1</u> is the full subcategory of <u>WOCC</u> of occurrence systems with the initial marking being a set.

Proposition 61. The underlying functor U: <u>DEC</u>→<u>WOCC1</u> has a right adjoint DEC: <u>WOCC1</u>→<u>DEC</u>. ⇨129◈

$$\begin{tikzcd}
\underline{\text{WOCC1}} \arrow[r, shift left, "\text{DEC}"] \arrow[r, phantom, "\wedge"] & \underline{\text{DEC}} \arrow[l, shift left, "\text{U}"]
\end{tikzcd}$$

Proposition 62. The functor pair U, DEC restricts to an adjunction between <u>DEC</u> and <u>OCC</u>.

$$\begin{tikzcd}
\underline{\text{DEC}} \arrow[r, shift left, "\text{U}"] \arrow[r, phantom, "\vee"] & \underline{\text{OCC}} \arrow[l, shift left, "\text{DEC}"]
\end{tikzcd}$$

Proof: The underlying occurrence system of any decorated occurrence system is safe. ♦

Thus, the composition OCC = U DEC WOCC is a functor from <u>FSYS1</u> to <u>OCC</u>. This is interesting, although we could not find an adjunction between the two categories. It is not easy to devise a functor from <u>FSYS</u> to <u>OCC</u>. It seems, that, in general, compositionality breaks down (permutations of places symbolising tokens on the same place). From the literature adjunctions are known for safe systems [Win84b], semi-weighted systems ([NS98], here post t must be a set for any transition t) and nets with restricted morphisms ([MMS92], here the images of the places in the postset of a transition must be disjoint).

Although our definition of a morphism is different, there are similarities:

- WOCC unfolds safe and semi-weighted systems to safe occurrence systems.
- OCC is a functor on general morphisms. The restriction concerns only the objects, namely, that the initial making must be a set. Although OCC is not part of an adjunction,

there is a morphisms ε_S: U OCC S→S which factorises any process of S (but not uniquely).

Our definition of decorated occurrence systems is a modification of that in [MMS92]. We number all output-places of a transition whereas [MMS92] enforces the order only on subsets corresponding to one output-place of a transition in the weighted net. This implies different restrictions and different functors. This method works also in our framework. But in [MMS92] an adjunction is achieved what we couldn't do here, again, because we disallow to map places to markings.

What is the difference between the semantics given by WOCC and OCC? First OCC generates a place for each single token whereas WOCC may use a place for several tokens. In the last section we called this radical individual tokens philosophy and suggested that WOCC might be more appropriate for anonymous tokens. This view is supported by the restrictions on the functor OCC: only a subcategory allows to individualise tokens compatible with morphisms. It is out of the scope of this paper to discuss parallels to quantum physics. But, it is remarkable that the borders of this adjunction to safe occurrence systems are so similar for different authors although the definition of morphisms are different. This might indicate that this border very deeply rooted in net theory.

Now, we may compare thre semantics of a system S, namely, WOCC S, OCC S and
PROC S = {o: O→S | o is a process of S}.

Proposition 63. The expressive power of WOCC is strictly stronger than that of OCC which again is strictly stronger than that of PROC.

$$\begin{tikzcd}
 & \textit{\textbf{Nets}} & \textit{\textbf{Systems}} & \textit{\textbf{Semantics}} \\
 & & \underline{\mathbf{FSYS1}} \arrow[r] \arrow[d] & \underline{\mathbf{OCC}} \\
\textit{\textbf{Folding}} & \underline{\mathbf{FNET}} \arrow[r, shift left] \arrow[r, phantom, "\parallel\ \nabla"] \arrow[d, shift left, "\mathbf{U}"] & \underline{\mathbf{FSYS}} \arrow[l, shift left] \arrow[r, shift right] \arrow[r, phantom, "\nabla\ \parallel"] \arrow[d, shift left, "\mathbf{U}"] & \underline{\mathbf{WOCC}} \arrow[l, shift right] \\
\textit{\textbf{Place-Preserving}} & \underline{\mathbf{PPNET}} \arrow[u, shift left, "\mathbf{F}"] \arrow[u, phantom, "\lhd"] \arrow[r, shift left] \arrow[r, phantom, "\parallel\ \nabla"] \arrow[d, shift left, "\mathbf{U}"] & \underline{\mathbf{PPSYS}} \arrow[u, shift left, "\mathbf{F}"] \arrow[u, phantom, "\lhd"] \arrow[l, shift left] \arrow[r] \arrow[d, shift left, "\mathbf{U}"] & \underline{\mathbf{SM}} \\
\textit{\textbf{Clustering}} & \underline{\mathbf{PTNET}} \arrow[u, shift left, "\mathbf{PP}"] \arrow[u, phantom, "\lhd"] \arrow[r, shift left] \arrow[r, phantom, "\parallel\ \nabla"] & \underline{\mathbf{PTSYS}} \arrow[u, shift left, "\mathbf{PP}"] \arrow[u, phantom, "\lhd"] \arrow[l, shift left] &
\end{tikzcd}$$

Figure 26. The synopsis of categories and adjunctions.

Proof. *Stronger* follows from OCC = U DEC WOCC and that every process of a system S factorises through ε_S: OCC S→S. The semantics differ on the examples of Figure 19. Together this yields *strictly stronger* ♦

[MMS94] compared process semantics and safe unfoldings. The finding that "the unfolding contains several copies of the same process which, ..., are needed to provide a fully causal explanation of the behaviour" fully applies to the current setting. But it applies also to the <u>WOCC</u> and <u>OCC</u> unfolding and one would expect that <u>WOCC</u> semantics lays between the process and <u>OCC</u> semantics. The same expectation comes from the discussion that WOCC individualises tokens only partially. But the previous proposition falsifies this expectation! The explanation is that WOCC remembers more about the distribution of tokens on nodes of the net (and similar of occurrences to transitions) - a phenomenon countering the above argumentation.

In conclusion, we got a web of categories, functors and adjunctions depicted in Figure 26. Clearly visible are

- the bridge from clustering to folding (<u>PTNET</u> over <u>PPNET</u> to <u>FNET</u>),
- the dichotomy of structure (<u>*NET</u>) and behaviour (<u>*SYS</u>)
- the connections to semantics (<u>SM</u>, <u>OCC</u> and <u>WOCC</u>)

This two-dimensional schema may be embedded in three dimensions with an additional axis from low-level to high-level nets, i.e. flat, coloured and hierarchical nets. This is shown in Figure 27. It suggest to interpret Figure 26 as the back and the colourings from section 3.5 as the floor of a cube. Naturally, the questions arises how the other walls should look like. For example which relationships does the process semantics of a coloured system possess? It is a interesting task for further research to complete this cube.

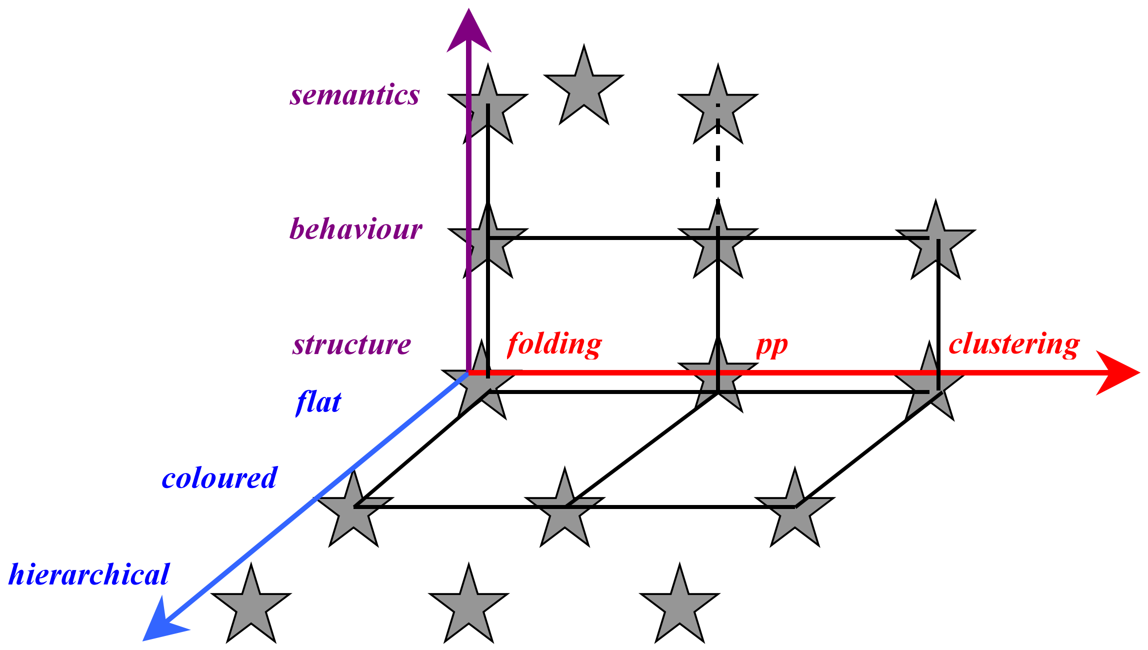


Figure 27. Categories, adjunctions and functors in a 3-dimensonal schema.

4 Reverse engineering

4.1 Introduction

Within the setting of this paper, the core task of reverse engineering may be stated as:

- for a given uncoloured net, find an elegantly structured high-level net which unfolds to the given net and explains its design

To apply this approach to a general system - not necessarily implemented in Petri nets - the following procedure is proposed

- Choose a translation method appropriate to the goal of the reverse-engineering task.
- Translate the existing system by the chosen method to a Petri net.
- Analyse the net in order to recover reductions, colourings, building components etc..
- Interpret the most promising analyses by translating them back to the original system. Decide which results to use, which not to use, which to refine and how to combine them.

As discussed in section 1.2, this chapter is mainly restricted to the subtask of recovering a coloured net. First, however, several methods of translating software systems to nets are presented. A specific example is also described. This is a spare-part order system which will be used as current example throughout this chapter. We consequently provide a much more detailed description for it than for the other translation methods.

The next subchapter gives the first rudimentary version of the reverse-engineering algorithm to be developed. It is rather surprising that this version already yields a reasonable analysis. The main purpose is to build the type system as the basic data structure and to introduce the prototype implementation in Smalltalk. Readers unfamiliar with Smalltalk may refer to Remark 67.

When the base classes have been built, we can introduce the core of the algorithm: computing a reduction by merging neighboured transitions - iteratively. This merely constitutes the implementation of the iterated universal constructions from the net-theoretical part. However, sophisticated techniques are required to get the algorithm to run correctly and quickly. The same applies to computing its cost. Ultimately, this algorithm is fast, stable and flexible. It turns out to be really easy to implement different neighbourhood definitions - a first proof of its flexibility.

The algorithm is enhanced by introducing heuristics, namely in relationship-analysis techniques for the current example. These are initially used to enhance the discriminating power of the algorithm so that it now differentiates between the nodes of the 1 and the n side of a reflexive 1:n relationship. They are then used to find the structure of a given reduction in terms of colourings. The recovered coloured net simultaneously contains the design specification and a no-loss representation of the implementation. In fact, reverse engineering reveals properties overlooked during the design phase.

The last section discusses variations, applicability and extendibility of this algorithm. Its quintessence is that the unchanged algorithm offers a good starting point for many reverse-engineering tasks. Furthermore, it can be integrated with domain-specific heuristics to produce deep insights. However, this step requires careful balancing of different factors.

4.2 Petri Nets as a Modelling Tool

This chapter discusses applications of Petri nets. It shows that both forward and reverse engineering may profit from Petri-net based methods, also in areas other than concurrency which is the traditional strength of Petri nets.

This requires a process of translation from the relevant application to a Petri-net model. It is merely a new kind of the modelling abstraction inherent in every engineering methodology. It subsequently turns out that there are different natural and intuitive translations of software systems to Petri nets.

The first folding-based Petri-net models will describe a spare-part order system. It is complex enough to illustrate and detect the strengths, weaknesses and potentials of the ideas expressed in this paper and will serve as a current example. The example is introduced by a series of Petri nets connected by morphisms. It illustrates how the categories introduced in the previous chapter can be applied to an engineering task. Moreover, the implementation of the Petri-net design in a procedural language using a relational database is explained in some depth. Later on, implementations or unfoldings of this system will be used as a test input to the reverse-engineering algorithms.

The second section uses a different translation to Petri nets which supports clustering-based engineering methods. The principles of structured programming will be modelled with Petri nets by making use of the clustering capabilities of Petri-net morphisms.

The program semantics will subsequently be translated as Petri nets. Two different approaches are discussed: explicit representation of the control flow or a pure data flow. The former method may be used for a fine-grained analysis which must preserve the semantics on a microscopic level whereas the latter could be used to migrate a sequential program into a parallel one, for instance.

The fourth section looks at reverse engineering without source code. For a Petri-net method, the switch to binary machine code or execution traces seems simpler than for methods working on the symbolic level.

As a final example we use an embedded system designed by a Petri-net tool. Because we already started from a net, a translation is not necessary. However, the net was generated by a tool using modularization concepts very different from ours. It will therefore be interesting to observe how the reverse-engineering algorithm reacts to this concept clash.

This chapter shows that the reverse-engineering methods to be developed in the rest of this work apply to very different tasks and that folding-based Petri-net methods constitute a valuable engineering metaphor - also for purely functional tasks which cannot benefit from the natural strengths of nets in concurrency.

4.2.1 Forward Engineering by Folding

Using morphisms for the design of Petri nets leads to the following translation of terminology:

- f: N→N'   N implements N'
- N' = im f   abstraction or design
- N = src f   specialisation or implementation
- $f^{-1}(x')$ implementation of node x'
- $f^{-1}(\mathbf{0})$ structure clash

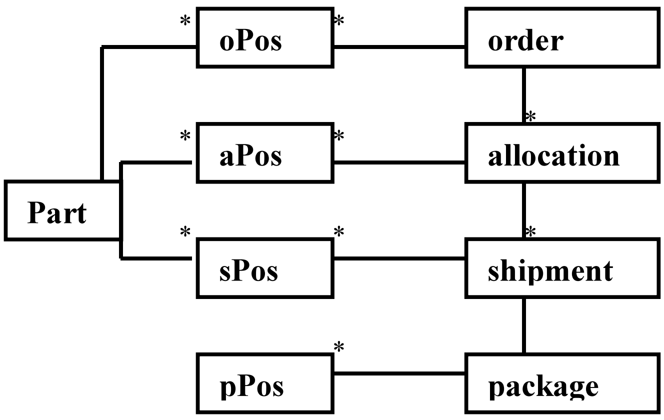


Figure 28. The class diagram of the spare-part order system.

We will describe an example system first by a class diagram and then by a series of nets. It is a spare-part order system for a big loom manufacturer in Switzerland. The author was involved in its implementation in a procedural 4GL some years ago.

Figure 28 is the class diagram of the system. An order consists of different positions each one containing a certain number of a spare part. Once the order has been defined, parts get allocated, i.e. reserved in the warehouse. If parts are out of stock or have to be manufactured specially an order can be split in several allocations. If an order contains old parts no longer in production then they are replaced by a compatible replacement part. For this (transitive) navigation in a complicated replacement graph is used. An allocation is distributed in possibly more than one shipment such as to allow the parts of a shipment can fit into one box.

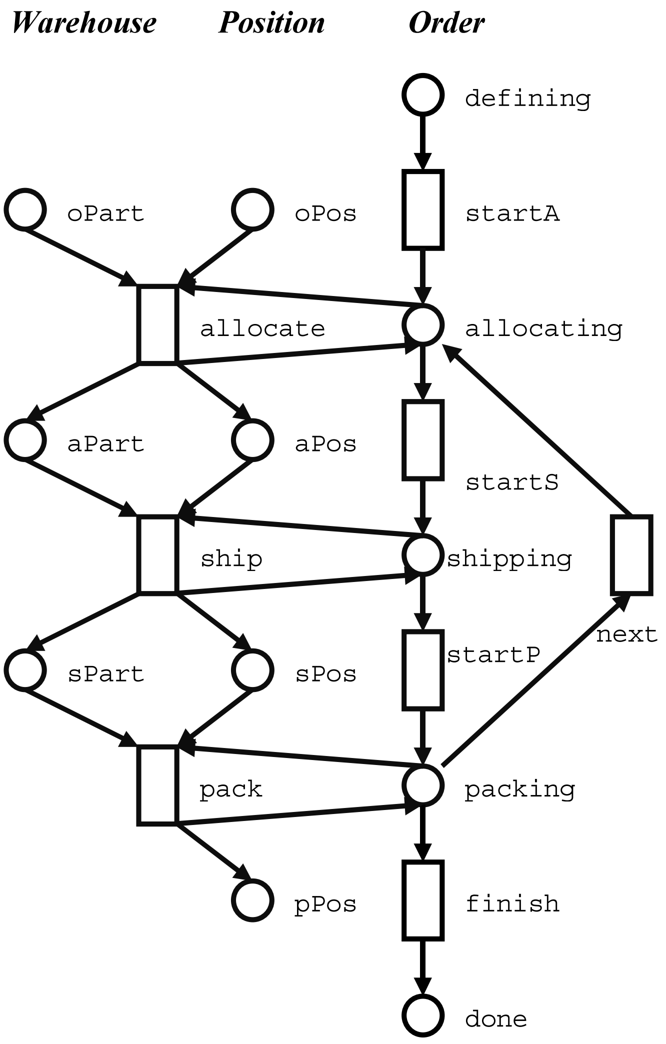


Figure 29. The design of the spare part system as the net N^u.

The dynamic model of the spare-part system is depicted in Figure 29 as an uncoloured Petri net. After the positions of an order are defined, they are allocated. An allocation is distributed into one or more shipments. The warehouse robot transports the parts of a shipment to the packing station, where they get packed into a box together with all necessary shipment documents.

Figure 29 shows

- vertically:
  - the life cycle of an order consisting of the states allocating, shipping and packing
  - the life cycle of a part waiting on stock in the warehouse, getting allocated, shipped and packed
- and horizontally:
  - the interface between an order and a part, they are connected in an interleaving way, enforcing, that a part changes its state only, if the order has the corresponding state.

The left column shows the handling of parts by the warehouse. A token signifies

- in `oPart`: an available part on stock,
- in `aPart`: a Part on stock allocated to an order and
- in `sPart`: the robot transports the part.

The places in the middle column correspondingly show the life cycle of an order position. And the right side reflects the state of an order.

This uncoloured net – let's call it N^u - deals with a single part. To handle several parts use a coloured net $C^{P \to u}$: $N^P \to N^u$ and a set PART to represent the different parts:

the nodes of N^P are
places: the cartesian product of the part-places {`oPart, aPart, sPart`} and the position places {`Pos, aPos, sPos, pPos`} with PART and the remaining places of N^u
transitions: the cartesian product of the part transitions {`allocate, ship, pack`} with PART and the remaining transitions of N^u
$C^{P \to u}$: $N^P \to N^u$ is given by C x = x and C (x, p) = x for any node x of N^u and part p ∈ PART
$pp^P (x_u, x_P) (y_u , y_P)$ = if $x_P = y_P$ then $(pp^u\ x_u\ (y_u))$ x $\{x_p\}$ else **0** else pp^u C x (C y)

This method is very simple it just multiplies nodes. In general, however, a design dimension contains additional application semantics. In our example we need to add the rules which find a replacement part for a part that is outdated or out of stock. For this the colouring of `allocate` becomes more sophisticated as follows:

{(`allocate`, oldPart, newPart) ∈ {`allocate`} x PART x PART |
newPart is currently sold and equals oldPart
or is a valid direct or indirect replacement of it}
pre^P (`allocate`, oldP, newP) = (`oPart`, oldP) + (`oPos`, oldP) + `allocating`
$post^P$ (`allocate`, oldP, newP) = (`aPart`, newP) + (`aPos`, newP) + `allocating`

This means the colours of the `allocate` transitions correspond to the transitive closure of the replacement relation. But the complexity of this closure and database normalisation are reasons to prefer alternative implementations.

To handle the further dimensions of orders, shipments and boxes we introduce the sets ORDER, SHIP and BOX that allow to uniquely identify a box of a shipment of an order etc.. The coloured net C: $N^{POSB} \rightarrow N^{u}$ is defined analogously to C^{P}. The details may be found in

Remark 64. Colouring of orders, shipments and boxes. ⇨130◈

Of course, a real system needs many more application dimensions and details. But for the purpose of this paper we stop the forward engineering process here and implement now. What does implement signify here? We could take the last Petri net already as an implementation. But to illustrate a translation of a Petri net to a procedural language we will reformulate the last net as a relational database application. A simple (abbreviated) Pascal- and -SQL-like syntax will be used.

First we define the tables and columns of the database. This can be done straightforwardly, using place names as table names and colours as attributes. The attribute `tokens` is used to hold the number of tokens:

```
   table oPart (part tokens)
   table aPart (part tokens)
   table sPart (part tokens)
   table allocating (order)
   table shipping (order ship)
   table packing (order ship box)
   table oPos (part order tokens)
   table aPos (part order tokens)
   table sPos (part order ship tokens)
   table pPos (part order ship box tokens)
```

The columns printed in bold together form the unique primary key for a table. The typing information is omitted because it is clear from the naming. The tables `allocating`, `shipping` and `packing` drop the `tokens` column because a tuple may only contain 0 or 1 token. Thus, existence in the database means one token non-existence means zero. Of course this database design can be improved in many ways but for the current purpose we choose this simple approach.

The transitions are translated into database transactions, programmed as parameterised procedures, let's start with a simple one:

```
PROCEDURE startS(anOrder aShip);
   BEGINTRANSACTION;
   DELETE allocating WHERE order = anOrder;
   INSERT INTO shipping (order, ship) VALUES (anOrder, aShip);
   COMMIT;
END startS
```

This procedure has two input parameters: `anOrder` and `aShip`. Again typing information is hidden in the naming. The procedure executes two SQL-statements first it deletes a tuple from allocating, then it inserts a tuple into shipping. We freely mix Pascal and SQL-Syntax avoiding name clashes by appropriate prefixing or renaming. Hidden in these SQL-statements is an existence condition: if no tuple of `allocating` fulfils the `WHERE`-condition the DELETE statement will fail. At our level of abstraction error handling is not specified, so this is a valid implementation of the existence condition. Similarly INSERT fails for the primary key constraint, if the uniqueness of `(order, ship)` is violated.

The SQL-statements correspond exactly to the pre- and post function of the net; the translation process is straightforward.

Now we see a consequence of the decision to drop the tokens attribute in order. Because `startA` deletes the order in the table allocating, it is no longer there to enforce by the primary key constraint the uniqueness of order. Hence additional logic is needed to ensure uniqueness when creating a new order. `startP` is similar:

```
PROCEDURE startP(anOrder; aShip; aBox);
   BEGINTRANSACTION;
   DELETE shipping WHERE order = anOrder AND ship = aShip;
   INSERT INTO packing (order, ship, box) VALUES (anOrder, aShip, aBox);
   COMMIT;
END startP
```

Algorithm 65. The procedures `next` and `allocate`. ⇨130◈

Although, these procedures are slightly more complicated the same principle guides the implementation. In conclusion, implementation is done by the following translation:

- places are translated into database tables
- the colours of a place give the columns of the table
- transitions are programmed as procedures
- transition colours give the procedure parameters
- pre- and post functions are reflected as database modifications

Still missing is the outer block code or the driver of these isolated procedures. In a Petri net the firing of transitions is driven by the paradigm that any enabled transition non-deterministically may or may not fire. For a procedural implementation the situation is different:

Remark 66. Main Driver. ⇨131◈

Using the colours P = PART, O = ORDER, S = SHIPMENT and B=BOX our example expands to the hierarchical net in Figure 30. Above the empty net **0** this diagram contains 2^4 nets of our example. The superscripts indicate the colours that are present in the net, so N^u is the uncoloured net at the beginning of the last chapter and N^{POSB} the last net with all the four colours. The morphisms (except 0) are π_c with c the colour, that is dropped from their source to their target. The forward engineering in the last chapter started with N^u and followed the left side of the diagram via N^P to N^{POSB}.

The diagram is commutative. Notice that in general a square is neither a pushout nor a pullback square. This means as well that in a single square as depicted in Figure 31 the two lower morphisms are not redundant. This is not true for the whole diagram. The 4 morphisms

$$\begin{tikzcd}
 & & N^{POSB} \arrow[dl] \arrow[d] \arrow[dr] \arrow[drr] & & & \\
 & N^{POS} \arrow[dl] \arrow[d] \arrow[drr] & N^{POB} \arrow[dll] \arrow[d] \arrow[drr] & N^{PSB} \arrow[dll] \arrow[dl] \arrow[drr] & N^{OSB} \arrow[dl] \arrow[d] \arrow[dr] & \\
N^{PO} \arrow[dr] \arrow[drr] & N^{PS} \arrow[d] \arrow[drr] & N^{PB} \arrow[dl] \arrow[drr] & N^{OS} \arrow[dl] \arrow[d] & N^{OB} \arrow[dll] \arrow[d] & N^{SB} \arrow[dll] \arrow[dl] \\
 & N^{P} \arrow[dr] & N^{O} \arrow[d] & N^{S} \arrow[dl] & N^{B} \arrow[dll] & \\
 & & N^{u} \arrow[d] & & & \\
 & & 0 & & &
\end{tikzcd}$$

Figure 30. The hierarchical net.

with source $\mathbf{N}^{POSB}$ determine the whole hierarchical net. But this is not a simple colimit construction. Rather these four top morphisms give the maximal resolution of $\mathbf{N}^{POSB}$: The constructions would be governed by a set of rules of the type:

- if $\pi^O x = \pi^O y$ and $\pi^S x = \pi^S y$ then $\pi^B \pi^P x = \pi^B \pi^P y$

Formalising such rules would give an exact meaning to the term independent colour sets. Immediately two questions arise:

- how to express such rules categorically
- how to use such a formalisation for reverse engineering.

$$\begin{tikzcd}
 & N^{PSB} \arrow[dl, "\pi^B"'] \arrow[dr, "\pi^P"] & \\
N^{PS} \arrow[dr, "\pi^P"'] & & N^{SB} \arrow[dl, "\pi^B"] \\
 & N_S &
\end{tikzcd}$$

Figure 31. A single square.

Both questions are beyond of the scope of this work.

4.2.2 Structured Programming and Clustering

A more conventional approach to software engineering composes a system of subsystems using a limited number of composition constructs. E.g. the structured programming constructors:

- sequence
- choice
- iteration

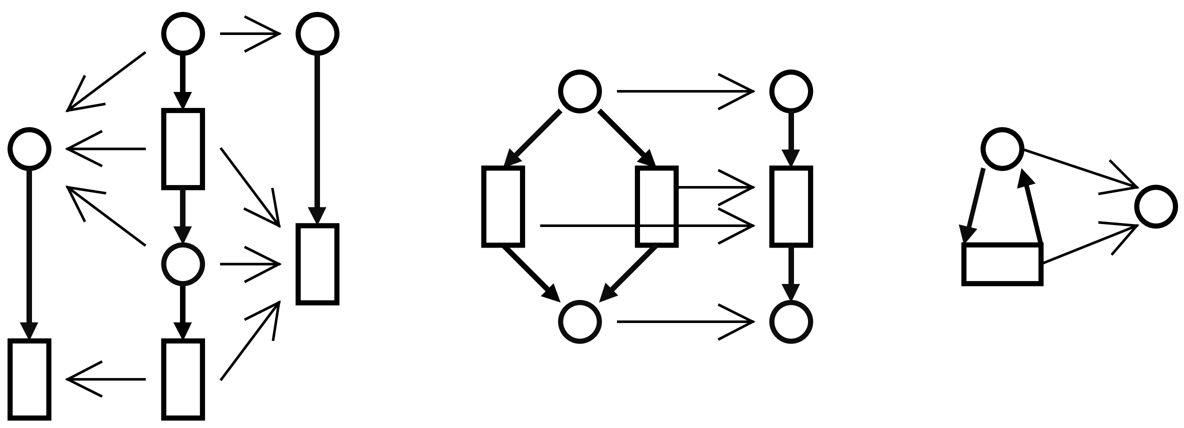


Figure 32. Structured programming: sequence (left), choice (middle) and iteration (right).

The reduction diagrams in Figure 32 clearly show that a net morphism easily reduces each of these constructs. It allows even the choice to map a sequence to a place or a transition. The net models naturally combine data- and control-flow together. So there is a chance to handle them symmetrically.

However, state of the art includes further constructs like

- parallel execution (fork)
- synchronisation (join)
- return or another atomic operation to leave a cascade of nested blocks
- call backs
- exception handling

Also fork and joins are nicely mapped, which is one of the natural strengths of Petri nets. However, the situation gets more difficult for the remaining constructs because they break locality. That is what they have been invented for – experience has shown that without such breaks it becomes cumbersome to localise functionality such as control loops or error handling. Hence it is difficult for any formalism to describe their semantics clearly.

For forward engineering, such a morphism together with a detailed description might be sufficient, but, how should we recover such a morphism in reverse engineering? Is there another alternative than to try to catch all clean constructs first, and let the user try to understand the rest?

Reverse engineering structured-programming constructs is a well-known technique. We work bottom up and replace every structured construct with a corresponding super place or super transition. For example there are tools that harness the transition from cyclomatic to essential complexity metric [Cab76], [Wat96] for this purpose. We sketch in the following the basic ideas as to how to apply this in our Petri-net approach.

For a given net N we need to compute a reduction f: N→N'. We will compose this morphism f by elementary reductions f_i:

$$N = N_n \xrightarrow{f_n} N_{n-1} \xrightarrow{f_{n-1}} \ldots\ldots \xrightarrow{f_2} N_1 \xrightarrow{f_1} N_0 = N'$$

Each of the f_i should eliminate a single construct. We will simply compose elementary reductions as long as we can find any. In Smalltalk (refer to Remark 67 for a basic introduction) this would look like:

```
Net>>clusterReduction
  | red |
  red := Morphism identityOn: self.
  [red needsFurtherReduction] whileTrue: [
    red := red image elementaryClusterReduction ° red].
  ^ red
```

To get an elementary reduction, we first look for the nice structures, and resort to locality breaking structures only if we are forced to:

```
Net>>elementaryClusterReduction
  | red |
  (red := self findMaximalSequenceReduction) isNil ifFalse: [^ red].
  (red := self findChoiceReduction) isNil ifFalse: [^ red].
  (red := self findIterationReduction) isNil ifFalse: [^ red].
  (red := self findForkJoinReduction) isNil ifFalse: [^ red].
  ^ self findLocalityBreakingReduction
```

Obviously, this is only a sketch of an algorithm, but we do not want to go in any details here.

4.2.3 Program Semantics

There are different visual programming environments that are implicitly or explicitly based on Petri nets. If the user wants to add two numbers, she or he selects an arithmetic add transition, drops it on the desktop, drags input edges from the places holding the numbers to add and finally draws an output edge to deposit the result in a further place.

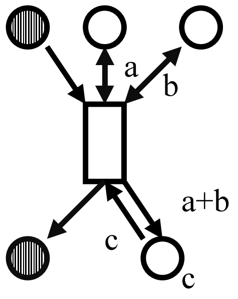


Figure 33. c := a + b with explicit control-flow.

Of course we can use the reverse technique to translate an existing system to a Petri net. But there are many choices how to translate. The next section shows a few of these possibilities in an informal way. The goal is to give a general idea, not to define algorithms.

A net for an assignment c := a + b with explicit control-flow is depicted in Figure 33. There is one place for each of the variables a, b and c. The places representing the control flow are filled with vertical lines.

A sequence of two blocks x and y is modelled as shown in Figure 34. x and y may be simple transitions or complicated components. Each of the component nets has one input and one output control flow place. A sequence simply merges the control flow output of x with the input of y. Choice and iteration are composed similarly.

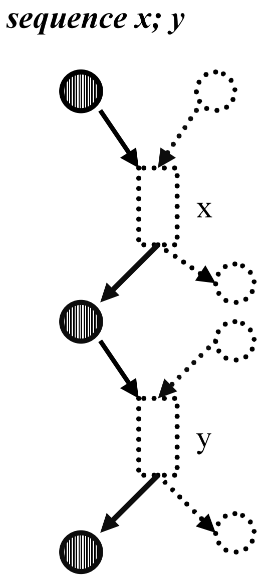


Figure 34. A sequence in explicit control-flow representation.

Additional work is needed for procedure calls. These for two reasons

- the return address has to be remembered
- for recursive calls, the local variables (including the parameters) need to be instantiated in a separate copy.

There are many ways to handle that in Petri nets and many high-level nets have a special construct for this functionality. For example CPN's ([Jen92]) use page instances.

We just give a simple solution. Every such procedure gets a place iCounter for an invocation counter, initialised with 0. The control flow places in the procedure hold instead of the anonymous black token the current ic. All the local variables become tuples (ic, val) where ic is the current invocation count and val the application value of the variable.

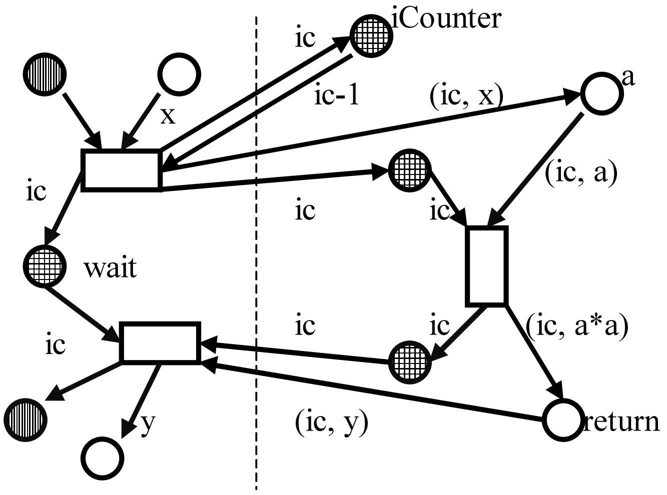


Figure 35. A caller (left) of the function a^2 (right).

If a user calls the procedure it fires a transition that commutes a new invocation count ic and passes the parameters together with the new ic to the input places of the procedure. Every transition in the procedure gets a guard condition which ensures that it uses only variable values with the correct ic. This is shown in Figure 35 in which the places holding invocation counters are chequered. The net for this simple call looks rather complicated but it contains the whole machinery of recursive procedure calls.

This translation with explicit control flow is a no loss conversion which contains the exact semantic of the source program. In fact it is a compiler which generates Petri nets.

An alternative approach is to drop control flow completely. This allows to get rid of the control flow places and the doubled arrows for reading and writing variables. This looks very intuitive as in Figure 36 – this is why visual programming languages use this kind of diagram, instead of the diagram in the last paragraph.

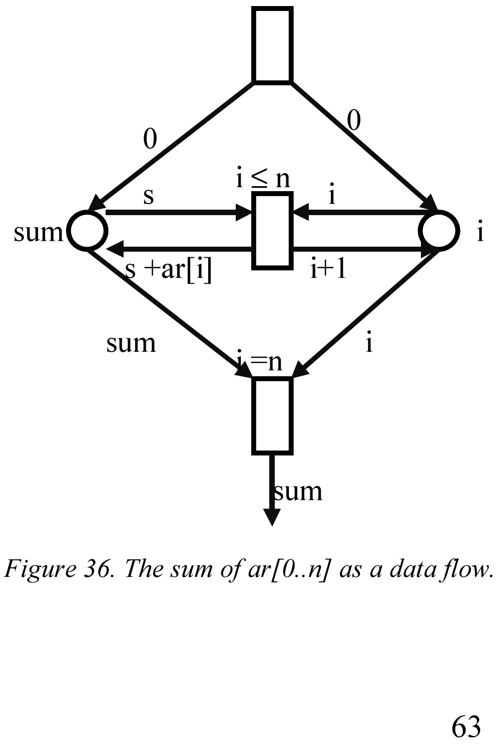


Figure 36. The sum of ar[0..n] as a data flow.

The net is no longer forced to the fixed sequential execution flow of the procedural program. The concurrent semantics of transition occurrence allows a highly parallel execution, only restricted by data availability. Hence translating a procedural program to a net without explicit control flow has the potential of transforming it into a parallel algorithm. But of course there are restrictions:

- a program is not only serialised by control flow restrictions data dependencies can do the same - as the loop example shows. To get a really parallel version of this loop we have to use a different algorithm which exploits associativity of addition. However, this is beyond the scope of a simplistic Petri net approach.
- we have to remove input tokens from the places otherwise the transitions will repeat occurring and reproduce the same output forever.
- if the same value should be read more than once from the same variable it has to be known in advance and the value has to be copied explicitly.
- If different values have to be written into a single place for example in a loop then it must be ensured that the old values are removed and that the consumer transitions realise when there are new input values.

There is of course a lot of research and literature about the conversion of sequential programs into parallel programs. We only want to mention that Petri net theory offers a various tools to attack these problems (e.g. [Fer94]):

- We can individualise the variables for each different use. In a high-level net, this can be done similarly as for the procedure calls in the previous section.
- We can use semantic preserving net transformations to remove control flow places. Thus a net with a mixture of explicit and implicit control flow is obtained.
- processes or partial order semantics of Petri nets individualise tokens. Hence by comparing (appropriate) processes of a net with explicit control flow and one without one can decide whether the data tokens are used in the correct sequence and the second net is a correct transformation of the first one.

4.2.4 Binary Reverse Engineering and Execution Traces

Also, there are also reverse-engineering techniques that do not operate on source code. The translations from the last section may be adapted to binary machine code or byte code. If the transformation with explicit control flow is used this yields a decompiler (on the lowest level, however, further analyses may reach higher levels of abstraction). The data flow approach could also be used for optimisation of existing code for parallel machines.

A further technique is to instrument the source code to collect trace information at selected points. The trace information can be analysed e.g. for performance questions. But, it may also be used for reverse engineering e.g. to relate a specific functionality with a code location etc..

The trace point can be selected in different ways:
- manually attached to the source code. It is one of the oldest debugging techniques to add the display statements (and to forget to remove them before migration to production ...)
- generated for example at every procedure entry/exit
- the interfaces to another software component
- at (nearly) regular time intervals

For reverse engineering there are different interesting applications possible:
- Many features are hard or impossible to extract from source code. The execution trace shows how it is puzzled together in specific instances.

- A preliminary analysis may be confined to code which is frequently used. This is likely to exclude things like error handling which often compromises the software structure.
- The access frequencies to persistent data can reveal the split between infrastructure and volatile data.
- A trace of the interfaces shows the black box behaviour of a piece of software – and may work even if the source code is unavailable

The reverse-engineering approach presented in this work can easily use execution traces. The only requirement is that they translate to a Petri net. Analysing this net may reveal interesting structures of the investigated system. Definitely this offers broad applicability to our approach. We confine us to describe one example: using the database trace of our running example from section 4.2.1.

The obvious way to translate a database trace to a net, is

- transaction $\rightarrow$ transition
- relation tuple $\rightarrow$ place

The crucial point is to derive a kind of tuple identity, such that a single tuple is mapped always to same place although its value may have changed. Database operations translate as

- insert $\rightarrow$ edge from transition to place
- update $\rightarrow$ edge from place to transition plus the reverse edge
- delete $\rightarrow$ edge from place to transition

This information is contained in every database audit. The translation to a net is straightforward: every database transaction translates to a transition which is connected to all modified tuples. Simply the tuple identity must be mapped to places. For the example system this translation yields the net N^{POSB} of Figure 30.

4.2.5 The Telephone Net

This paragraph presents the last example domain to apply our reverse-engineering approach. It is the most obvious: the analysis of an existing Petri net. As an example we have chosen a

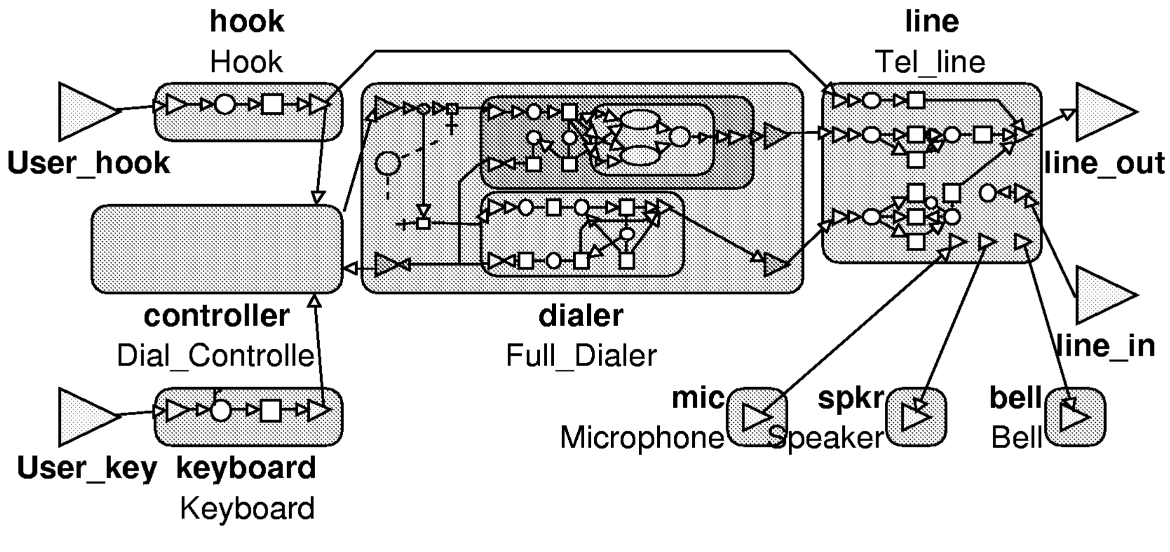


Figure 37. The telephone net implemented as a composition of CodeSign components.

net modelling a telephone. The net is delivered together with the CodeSign tool (see [Ess97]). A net is recursively composed of components. A component is in itself a (composed) net together with a set of connectors which may connect to net nodes or other connectors. They allow a restricted kind of multiplying tokens (send copies to several destinations). The telephone net is thus structured in a way that is very different and even incompatible with our approach. To reverse engineer such a net with our tools is hence a hard crosscheck.

Figure 37 shows the telephone net. The tool draws a diagram with many very small components. Some of them contain more interfaces (triangles), than internal nodes. A further complication is that the telephone net uses state machines. They are implemented as a user defined embedded formalism in CodeSign and are displayed as user defined symbols (ovals). CodeSign translates them internally into Petri net elements. Our algorithm will analyse the underlying place-transition net in which all components and state machines are resolved as places and transitions.

4.3 Simple Reductions

The previous sections presented a range of methods for modelling systems with Petri nets and discussed applications for the reverse engineering of these nets. This section starts with the core of this work, namely the development of a reverse-engineering algorithm for an unstructured net.

Reductions are morphisms from the net under investigation to a simpler net. They can give an extremely concise model of a net, although this must naturally disregard many details. Techniques are available to reintroduce the lost information such as node colourings or arc inscriptions. Performance is a very attractive feature from the point of view of reverse engineering: combinatorial explosions are naturally avoided. However, there is a danger of losing too much information. In the extreme case, a net consisting of a single transition and a single place is produced – hardly a helpful abstraction.

Looking back to section 3.5.3, a basic implementation is described for a maximal ι reduction with ι as a set of pairs of local injections. This chapter presents a basic form of this procedure. However, this simple form can already perform useful reverse-engineering work.

In this section, a type system is initially built. Single-transition subnets are used as basic types. The necessary data structures and algorithms are discussed. Small types are then composed to obtain larger types, thus producing a network of groups of isomorphic subnets connected by inclusions.

The third section shows that such a type system translates directly to a classification of the nodes of the net. Finally, this classification is used to build a reduction using a feedback technique. The feedback determines which types should be expanded and a new reduction is then computed with this expanded type system. Such a reduction algorithm already contains interesting information.

The prototype of the algorithms was implemented in Smalltalk. Most code examples are thus given in this language. A short summary of Smalltalk constructs used may be found in

Remark 67. Used Smalltalk constructs. ⇨132◈

4.3.1 Seed Types

The type system should classify subnets and isomorphic subnets should have the same type:

Definition 68. The types of net N are the equivalence classes of a relation R over a set Σ of subnets of N which is closed under monomorphisms, i.e.

$\forall S \in \Sigma, \forall \iota: S \to N$ with ι monomorphic holds $S\ R\ \iota(S)$

First seed types are computed which are later combined into bigger types. A natural choice for the seed types are the subnets consisting of a single transition with the surrounding places.

For the algorithm it is not sufficient to look at a type as an equivalence class of subnets. Additionally the connecting monomorphisms ι from Definition 68 are necessary. One way to reach this is to store for each type C

- a disjoint copy N_C of one of its subnets,
- exactly one monomorphism from N_C to each of its subnets
- the automorphisms of N_C

This is shown in the class diagram Figure 38. Because `NetType` is a subclass of `Net` it is in itself a net, namely N_C. Thus it does not need an instance variable for N_C.

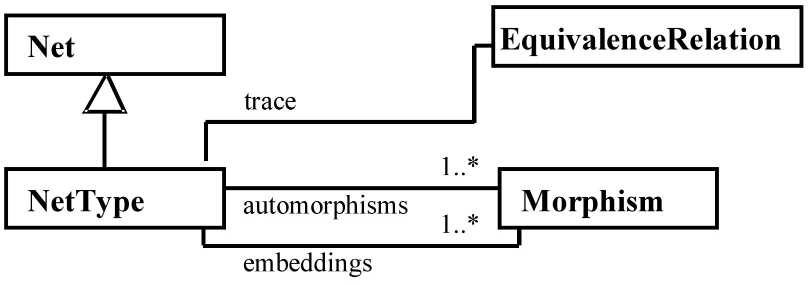


Figure 38. The class diagram for NetType.

In the equivalence relation trace, two nodes x and x' of N_C are equivalent iff there is an automorphism of N_C that maps x to x'. I.e. the equivalence class of a node x is the trace of x under the automorphisms.

A place p of a net with a single transition t may be characterised by a tuple of integers

- pp(t, p) = ((pre t) (p), (post t) (p))

This pp is the evaluation of the function pp of Lemma 22 at place p. The number of arguments indicates which definition must be used. Anyway the two definitions are in 1 to 1 correspondence. A whole single transition net maps to a sequence of integer tuples

- (pp(t, p_1), pp(t, p_2), ..., pp(t, p_n))

where p_1, p_2, ..., p_n are the places of the net. The computation of this representation for every transition of a net is sufficient for an efficient computation of the seed types:

- if the list is sorted (in an arbitrary but fixed way) it is a unique key for the type of transition. I.e. single-transition nets are isomorphic iff their sorted list representations are equal.
- The pp(x) = pp(x') generates an equivalence relation on the nodes of the net, which is the trace equivalence.
- a permutation of the nodes is an automorphism, iff each node is mapped to a node in the same trace class
- the group of automorphisms of the net is the cartesian product of the permutations of elements of the trace classes

The algorithm for the seed types is a straightforward application of these properties. It finds the type of a transition in a hash table with the sorted list of pairs as key. Of course it may turn out to be a longer programming exercise, depending on the available class libraries.

Definition 69. A net N has the small transition property iff for each transition $t \in T$ holds $\Gamma \geq \sum_{p \in P} \mathrm{pre}(t,p) + \mathrm{post}(t,p)$ for a (global) integer constant Γ.

For complexity question the small transition property is always tacitly assumed. For the applications we are studying here transitions with a high number of arcs do not seem reasonable. Even if the small transition property does not hold cost may remain reasonable, but do not elaborate on this.

Lemma 70. The seed types may be computed with cost O(e) with e the number of arcs of the net N. ⇨135◈

4.3.2 Combining Types

If the types are combined into bigger types appropriate subnets must be selected. As we concentrate on reductions by local injections it is natural to study subnets that cannot be reduced by local injections. Formally in terms of section 3.5.3:

Definition 71. An li-component of net N is a place-bordered subnet isomorphic with its maximal ι reduction with ι pairs of parallel local injections in fixpoint relation.

For the rest of this section ι is used in the same way as in the above definition. A type system using li-components is well suited for folding based reductions. On single transition nets local injections are simply monomorphisms, thus the seed types from the last paragraph are the correct starting points. The algorithm will use a 'forbidden zone' around each transition to test whether a subnet is a li-component:

Lemma 72. A subnet N' of N is a li-component iff each transitions t' of N' is disjoint from forbidden(t') with

forbidden(t') = $\{t \in T \mid t \neq t'$ and $\exists\, (f, f') \in \iota$ with im f = env t and im f' = env t'$\}$.

Proof: A direct consequence of Proposition 36 ♦

This lemma yields an efficient test whether the union of two li-components is itself a li-component: only the union of the forbidden transitions must be computed. Furthermore if the union of T' and the forbidden transitions equals T, N' is a maximal li-component of N.

But deciding whether the union of two components is local-injective is only one step to create a new type. Regarding Definition 68 all isomorphic subnets must be found and in regard to the class diagram of NetType all automorphisms. To record all this information an additional class `NetStructure` is needed, as shown in Figure 39.

Algorithm 73. The combination of two types. ⇨135◈

This algorithm computes the data for a combination of two types. Performance cannot be good because the problem is NP-complete in the size of the combined types. Hence types of big sizes must be avoided. One method is to create the single transition types, and only expand types when an analysis does really need it. When the analysis finds problems, it calls the method in

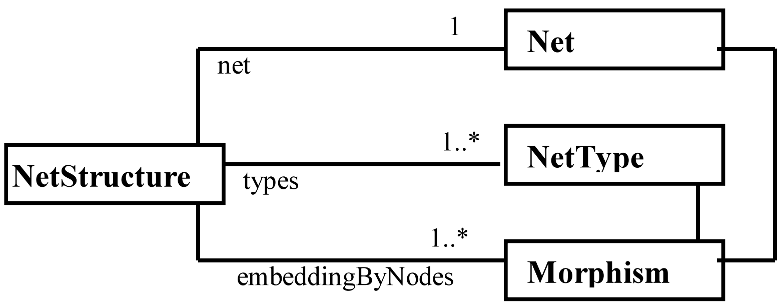


Figure 39. The class diagram for NetStructure.

Algorithm 74. Expanding problematic types. ⇨136◈

This method expands the types around the problematic zone by one transition after the other until the type system has effectively grown. Then the analysis is repeated for the larger set of types, which is hoped to reduce the problematic spots.

4.3.3 Classifying the Nodes

The concept of how to classify the node of the net is simpler to program than to formulate abstractly. In fact the idea was born, as a preliminary programming shortcut for a much more complex concept. First each node is classified by the types it is touched by. More precisely by the 'roles' it plays in each type. This is coded like:

Algorithm 75. Classification by type membership.

```
NetStructure>>membershipForTypes
  | byNode |
  byNode := IdentityDictionary new.
  self types do: [:type |
    type embeddings do: [:emb |
      type autos do: [:auto |
        type nodes do: [:node | (byNode
          at: (emb at: (auto at: node))
          ifAbsentPut: [self newSet]) add: node]]]].
^ byNode
```

Programmatically the role of a node n in a type is a node of this type that is mapped by a type embedding to n. This assumes the nodes of the types are disjoint. The only technical difficulty is hidden in the strange construct `self newSet`. This creates an empty instance of a special set class that supports equality based on the members contained in the set. In contrast equality of standard Smalltalk sets compares the object identity of the set instances. An optimised version of this method is contained in the appendix ⇨137◈ ♦

A node is classified not only by its types but also by the 'roles' it plays within each type. But it is ignored as to whether a node occurs only once or several times in a type embedding. This will be changed later to analyse relationships. A final technical preparation is needed before a net reduction may be computed:

Algorithm 76. Convert membership classification to an equivalence relation. ➪137◈

4.3.4 Reduction Ignoring Neighbourhood

We identified how to classify the nodes of net by types. In this section we will discuss a simple reduction algorithm which ignores the criteria neighbourhood and choice. It works as follows:

Algorithm 77. Net reduction by type expansion.

(i) Choose a net.
(ii) Convert the net to the internal format.
(iii) Compute transition seed types as in section 4.3.1.
(iv) Classify the nodes of net by the types (Algorithm 76) and transfer this classification into an equivalence relation (Algorithm 75).
(v) Build a net with nodes corresponding to the equivalence classes.
(vi) Check if the map from a node to its equivalence class is a morphism. If so the algorithm finishes with this reduction.
(vii) If not, the types touching the transitions not mapped as required by morphism, are extended until new types are generated by Algorithm 74.
(viii) If new types are found the algorithm loops back to step (iv)to reiterate with the expanded type system, otherwise it fails ♦.

The input for step (v) is an equivalence relation R. Create a new Net N' with one node for each equivalence class. Take the projection π: X→X' mapping a node x of N to the node x' in X' corresponding to the equivalence class of x in R. Define N' by

- $\pi\, x = [x]_R$
- $\pi\, x \in T'$ iff $x \in T$
- $pp'(\pi\, t, \pi\, p) = \sum_{\pi q = \pi p} pp(t,q)$

The second point is well defined because an equivalence class of R either only contains transitions or places only. The reason for that being is that the used type embeddings are foldings. The definition of pp' fails if $\pi\, t_1 = \pi\, t_2$ and the sums over the pp do not equal. Thus step (vi) computes the set of

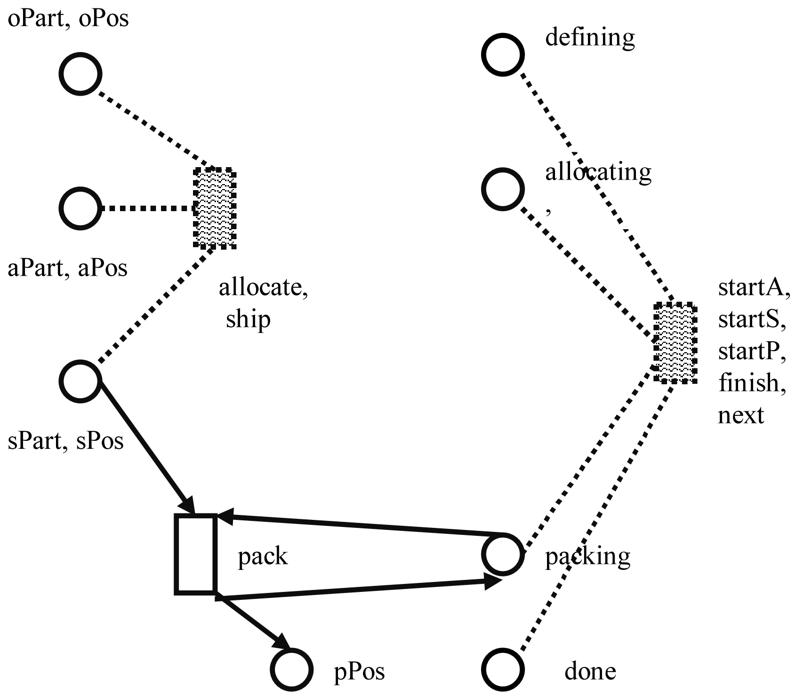


Figure 40. The reduction from the first iteration.

type embeddings with a destination transition at which pp' is not well defined. This set is the input for step (vii).

Now this algorithm may analyse the spare part net in Figure 29. We generated unfolded nets with several hundred nodes, with random distributions of part and positions of an order and random permutations on node and arc sequences.

Step (v) of the analysis algorithm produces at the first iteration the reduction in Figure 40. The conflicting transitions and their arcs are drawn in dotted lines. The node inscriptions show which nodes of the coloured net are mapped. Obviously this analysis is very basic, and directly shows the single transition types of the net.

At the second iteration step (v) finds a more interesting reduction which is shown in Figure 41. Step (vi) accepts this reduction as a valid morphism.

The major difference to the design Figure 29 is the additional column at the left side with names ending in a 1. These are the `allocate`, `ship` and `pack` transitions, that use a part occurring exactly once in a single position. Because these parts are 1 to 1 with the corresponding positions, there are no differences between parts and positions in reverse engineering. Hence parts and positions at the same stage are folded together.

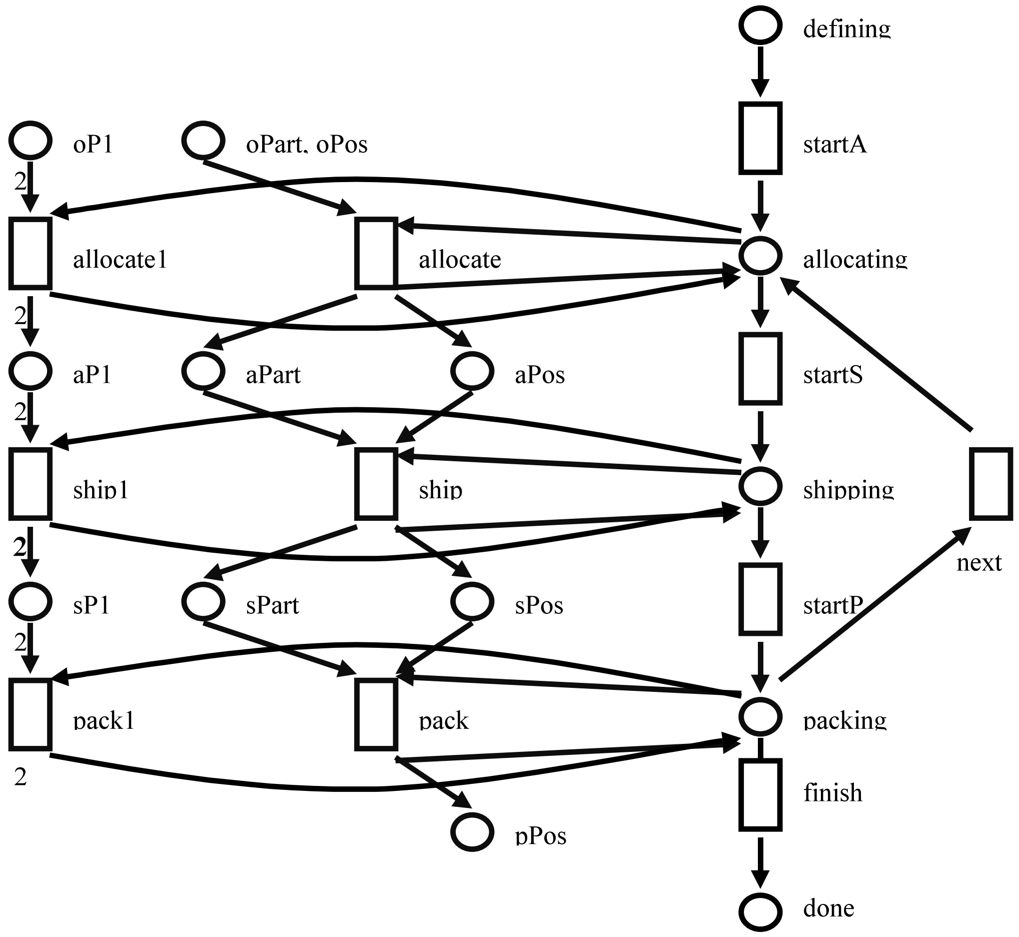


Figure 41. The reduction from the second iteration.

All transitions are separated, exactly as designed in section 4.2.1. But this is a happy accident due to simple structure of the example.

More surprising, the algorithm differentiates `aPart` from `aPos` and `sPart` from `sPos` instead of merging them such as `oPart` and `oPos`. In fact we first looked for a programming error to explain this... But the real reason is that the algorithm identifies a type as depicted in Figure 42. In this type `aPart x` cannot be mapped by a automorphism to `aPos y` or `aPos z`, hence they are not related in the equivalence relation. This logic does not apply to `oPart` and `oPos` because there is no second transition there.

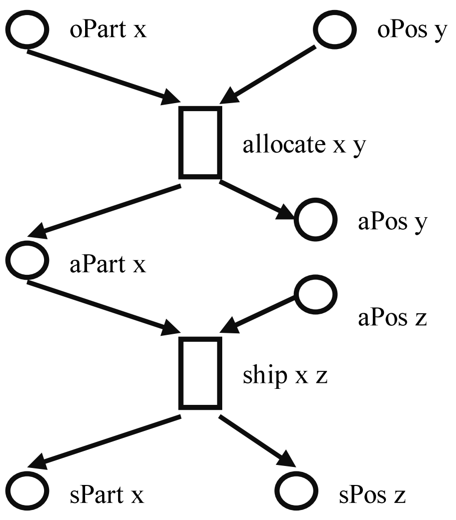


Figure 42. The asymmetry between parts and positions.

The complexity of the algorithm is optimal (i.e. O(e)) as long as it does not extensively expand types. But few type expansions imply a coarse relation and a small reduced net. So if the algorithm is useful because it detects high-level structures it is fast.

Summarising this simple algorithm detects valuable design structures and it has a natural stopping criterion. But there are two weaknesses:

- there is no guarantee, that it terminates (of course we can make it terminate, by keeping transitions separated, if the maps of their pre and post places do not correspond, but then we would loose the natural termination criterion)
- every iteration round results in a bigger net. We do not see a sensible method to get rid of the left column of 1 to 1 elements.

4.4 Neighbourhood Reductions

The last section computed basic data structures for the type system and a first reduction. This chapter describes the computation of a maximal reduction under consideration of a neighbourhood criterion. Thus subnets are merged if they are of the same type and, additionally, if they are adjacent. Of course, neighbourhood is used transitively: if the images of two subnets become neighboured in an intermediate reduction, they may get merged.

To get a fast algorithm - nearly linear in the input size - a combination of principles is used:

- work with single transition subnets according to Proposition 36.
- work locally
- work in the (intermediate) reduction not in the original net whenever possible
- avoid redoing things
- efficient data structures

Although this list looks rather obvious, quite a few non-trivial techniques must be carefully co-ordinated already at a theoretical level. The prototype implementation in Smalltalk revealed another set of problems. E.g. algorithms in the standard class library showed not the expected functionality or performance and consequently forced time-consuming re-implementations.

The first section outlines the algorithm and its variation for different adjacency criteria. In the second section it is used to analyse the spare part system. The results give a basic picture of the design although they are not yet a major improvement over the simple reduction from the last chapter. But it is the basis for the exciting results in the next chapter.

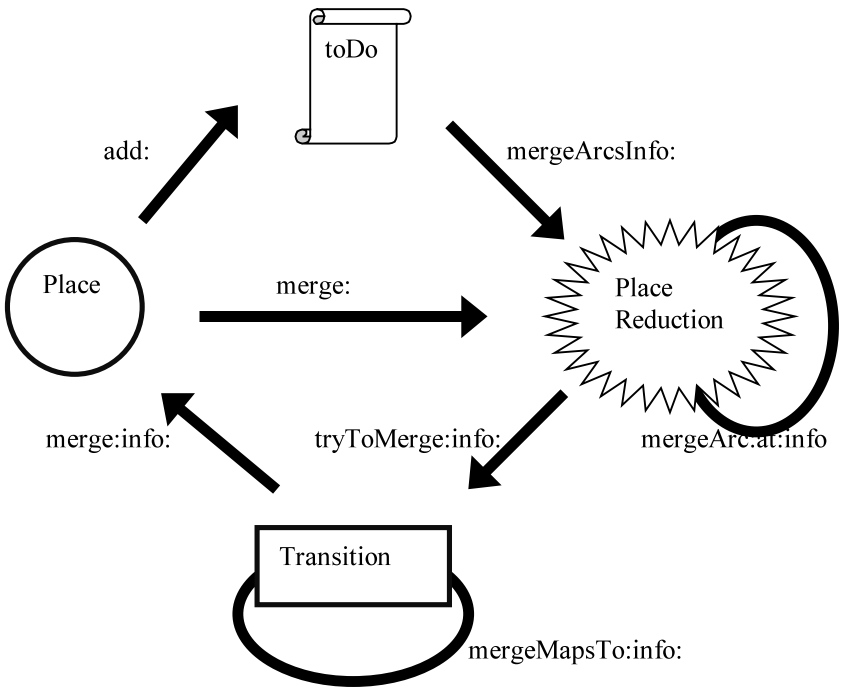


Figure 43. The reduction algorithm as object interaction diagram.

An attractive feature of the algorithm is its speed; it is nearly linear in the size of the input net which would be optimal. This is proved in the last section.

4.4.1 Algorithm

The main idea of the algorithm is to check the areas around a place in the intermediate reduction. If the origins of these arcs are connected to compatible transitions, they are merged which requires checking the arcs around these places recursively.

To obtain a fast algorithm it is essential, to minimise the number of times an arc is merged, because it's place has been newly merged. Therefore, the intermediate nets are not stored explicitly. Instead, an equivalence relation on the source nodes is used. The equivalence classes of this relation correspond to the nodes of the intermediate net in each step in the algorithm. Equivalence classes of places are mapped to instances of the class `PlaceReduction` which contains a representation of both the merged and the still to become merged arcs (around the reduced place).

The algorithm is represented in Figure 43 as a circular diagram. It should be read as: an instance of the class place sends the message `merge:` to an instance of `PlaceReduction` and a message `add:` to the `toDo` list. In the following each of these methods are explained in detail.

But foremost some preparatory remarks. The algorithm uses an object called info, which stores the information used during processing. It contains

- `relation`: an equivalence relation indicating, which original nodes might be mapped (if they get neighboured in the reduction). This relation is an input to the algorithm which does not change

- reduce: a map from the nodes of the origin net to equivalence classes. These correspond to the nodes in the current reduction. The equivalence classes of places are mapped to instances of PlaceReduction.
- toDo: a collection of the place reductions that still need to be merged

The initialisation builds the map reduce. It corresponds to an isomorphism and the representation of the arcs in the place reductions. Our implementation reuses code from the main algorithm for initialisation. This leads to an overlap of initialisation and main algorithm with a nice side effect: the toDo list is build implicitly. We spare the reader from the technical details of initialisation and termination and move directly to the main loop:

Algorithm 78. The main loop.

```
NetReduction>>mergeInfo: info
  self initialise
  [info toDo isEmpty] whileFalse: [
    (info reduce at: toDo removeAny) mergeArcsInfo: info].
  self finalise "translate it to a net and a morphism" ♦
```

This main loop is really straightforward: an arbitrary element from toDo is removed and the arcs of its current place reduction are merged as follows:

Algorithm 79. Merge the arcs of a place reduction.

```
PlaceReduction>>mergeArcsInfo: info
  | list |
  list := self nodesToMerge.
  self nodesToMerge: OrderedCollection new.
  list do: [:aNode |
    aNode keysAndArcsTreeDo: [:key :arc |
      self mergeArc: arc at: key info: info]]
```

The instances of PlaceReduction form trees. The root of a tree is the actual reduction and its other nodes are further reductions. The latter are already combined with the root but their arcs have not yet been merged. This tree structure allows the postponement of the merging of arcs and hence to minimise the repetitive merging of the same arc. Each node registers its direct children in the instance variable nodesToMerge. At the beginning of the method the current tree is saved in the variable list and then reduced to the root alone. Thus coping with the case that a reduction gets merged, while running mergeArcsInfo: ♦

keysAndArcsTreeDo: iterates through the tree and executes the code block for each arc to merge within the tree:

Algorithm 80. Depth first tree traversal.

```
PlaceReduction>>keysAndArcsTreeDo: aBlock
  self arcsByKey keysAndValuesDo: [:key :arcColl |
    arcColl do: [:arc | aBlock value: key value: arc]].
  self nodesToMerge do: [:node | node keysAndArcsKeyDo: aBlock].
```

A `PlaceReduction` registers the already merged arcs in a map called `arcsByKey` which maps keys to arcs. This is another key principle to obtain a fast algorithm. It avoids a linear search for a compatible arc which would cost somewhat quadratic in the number of arcs of a place. This is not acceptable because the small transition assumption (Definition 69) does not limit the arcs at a place. On the contrary in typical nets there are places with a huge number of neighbours.

What are the requirements for such a key? The most elegant requirement of course is

- arcs can be merged **iff** they have the same key

Then it suffices to check the keys. However, we will use the following weaker property:

- if arcs can be merged then they have the same key

This weaker property requires a compatibility check after the key check.

The main part of the algorithm only uses the keys and relies only on the above property. But the keys have to be constructed in the initialisation phase.

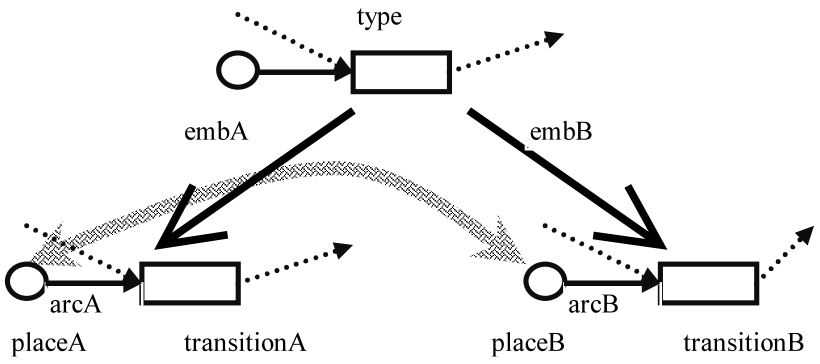


Figure 44. Merging two arcs.

What is the requirement to merge two arcs? In Figure 44 after the merger of `placeA` and `placeB`, `arcA` and `arcB` can get merged if

- both transitions have the same transition type called type
- there are two embeddings `embA: type` $\rightarrow$ `transitionA, embB: type` $\rightarrow$ `transitionB` and a place p of the type
- so that `embA(p) = placeA` and `embB(p) = placeB`

Thus an arbitrary embedding `embA` from type to the transition of `arcA` and all embeddings `embB` must be checked. But the restriction to an arbitrary embedding `embB` is sufficient, if trace classes (of places under automorphisms of the type see 4.3.1 Seed Types) are used. The reduced condition is

- there is a transition type `type`
- for arbitrary embeddings `embA: type` $\rightarrow$ `transitionA, embB: type` $\rightarrow$ `transitionB`
- the trace class of `embA`$^{-1}$`(placeA)` equals the trace class of `embB`$^{-1}$`(placeB)`

This means that the trace class is exactly the required key and merging arcs works as

Algorithm 81. Merge a single arc.

```
PlaceReduction>>mergeArc: newArc at: aKey info: info
   (self arcsAt: aKey)
     detect: [:myArc | myArc transition
        tryToMerge: newArc transition info: info]
     ifNone: [(self arcsAt: aKey) add: newArc]
```

As only the weaker key property is used `PlaceReduction>>arcsAt:` answers a collection of arcs (which in the reduction are connected to the receiving place reduction). If no such arc exists an empty collection is created by lazy initialisation. If it is possible to merge two transitions `Transition>>tryToMerge:info:` does it, returns `true` and `mergeArcAt:info:` finishes (by the Smalltalk semantics of `detect:ifNone:`). If the new arc cannot be merged with any of the existing ones then it is added to the arc collection. This is not necessary in the first case because the image the existing and the new arc coincide after the transition merger ♦

Algorithm 82. Merge two transitions.

```
Transition>>tryToMerge: aTrans info: info
  | maps |

  (info reduce relation is: self equivalent: aTrans) ifTrue: [^ true].
  (info relation is: self equivalent: aTrans) ifFalse: [^ false].
  (maps := self mergeMapsTo: aTrans info: info) isEmpty
    ifTrue: [^ false].
  info reduce relate: self to: aTrans.
  maps do: [:one |
    one keysAndValuesDo: [:ori :ima | ori merge: ima info: info]].
  ^ true
```

This method attempts to merge the receiver with the transition `aTrans`. If it succeeds or the transitions are already merged it answers true. A merger is rejected, if the two transitions are not compatible by the input equivalence relation (`info relation`). If there are merging maps the method applies to all of them to merge the environment of the two transitions ♦

The computation of the merging maps in

Algorithm 83. Merging Maps. ⇨138◈

is illustrated in Figure 45. The method composes each automorphism `auto` with `myEmb` and checks whether it is compatible with `othEmb`. Compatible means:

- mapped nodes are equivalent in the input relation
- map has fixpoints with regard to the current reduction.

This is the point of the algorithm to be varied for different adjacency criteria. In fact all four variations from **Definition 37** are easily handled as is explained in

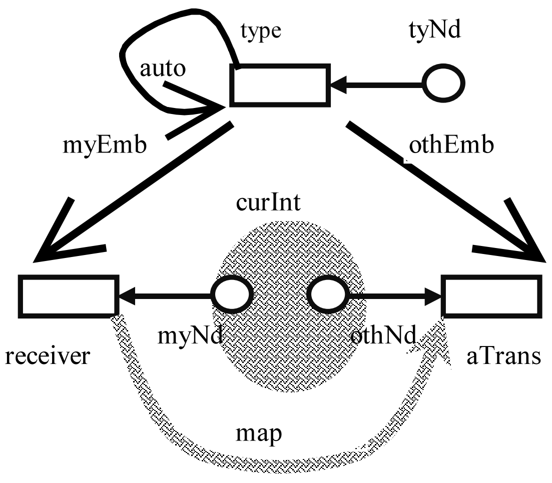


Figure 45. The computation of merging maps.

Algorithm 84. Adapting to different adjacency criteria. ⇨138◈

The effect of these variations cannot be demonstrated in the current example. All produce the same reduction because the example is too well behaved. It is left to further research to evaluate the implication of such variations in real-world programs.

The above method or one of its variations respectively is called from `tryToMerge` that uses the computed maps to merge the places in the environment of the transitions:

Algorithm 85. Merging two places.

```
Place>>merge: aPlace info: info
  | myReduction |
  (info reduce relation is: self equivalent: aPlace) ifTrue: [^ self].
  ((myReduction := info reduce at: self)
      merge: (info reduce at: aPlace)) == myReduction
    ifTrue: [info reduce relate: self to: aPlace]
    ifFalse: [info reduce relate: aPlace to: self].
  info toDo add: self
```

If the places are not already merged then the place reductions and the places are merged (in the equivalence relation `info reduce`). But the implementation requires these two mergers to be done in sync. This is the technical reason for the mathematically redundant branching. The merged places are added to the work list `toDo` which closes the circle of the algorithm ♦

The merging two place reductions requires linking one reduction into the tree of the other one:

Algorithm 86. Merging of two place reductions.

```
PlaceReduction>>merge: aReduction
  self arcsByKey size >= aReduction arcsByKey size
    ifTrue: [self nodesToMerge add: aReduction. ^ self]
    ifFalse: [aReduction nodesToMerge add: self. ^ aReduction]  ♦
```

The bigger `arcsByKey` map is reused. This optimisation will be crucial in the performance estimation.

4.4.2 Analysing the Spare-Part System

We used the spare-part system from Figure 29 to test our implementation. We used unfolded nets of different sizes as explained there. The algorithm needs an equivalence relation that allows or disallows the merging of nodes. For this, we used the type equivalence from section 4.3.3. The algorithm produced the analysis in Figure 46. Compared to the forward-engineering diagram in Figure 29 we lost

- The separation of parts and positions
- the colouring.

It is not surprising then that the algorithm cannot distinguish parts and positions. For example swapping `aPart` and `aPos` is an automorphism of `ship` or `allocate`. In fact, the surprise was that the algorithm using types in section 4.3.4 was able to differentiate them.

4.4.3 Complexity

Proposition 87. The reduction algorithm has in the worst-case cost of
O(e log (min (maxDeg(N'), | P |) γ)) with

- e the number of edges in the source net,
- maxDeg(N') the maximum number of arcs incident to a place of the reduced net,
- log the binary logarithm and
- γ a slowly increasing inverse of the Ackermann function, which does not surpass 3 in any real case

⇨139◈

A large maxDeg(N') normally does not correlate with a good design. Thus, if the algorithm is doing useful reverse engineering the cost is almost linear in e.

The processor time used by the Smalltalk prototype to analyse the running example in different sizes is shown in Figure 47. It does not look as linear as the above proposition suggests. This has several explanations:

- there are statistical variations, as Windows NT and VisualAge Smalltalk is are complex systems with many parameters not easily controlled
- nobody knows when garbage collection is running
- the implementation is not memory efficient, already with 40000 arcs is nearing the thrashing point. It's wonderful how much memory modern times software can use...
- tuning is a lot of work. It is not easy to localise the non linear things when approaching the thrashing point
- a lot of standard Smalltalk hash tables are used in dictionaries, sets and so on. If a program uses systematically distorted hash distributions the hash tables become very, very slow. This happened several times. But even if it is detected it is still time consuming to repair.

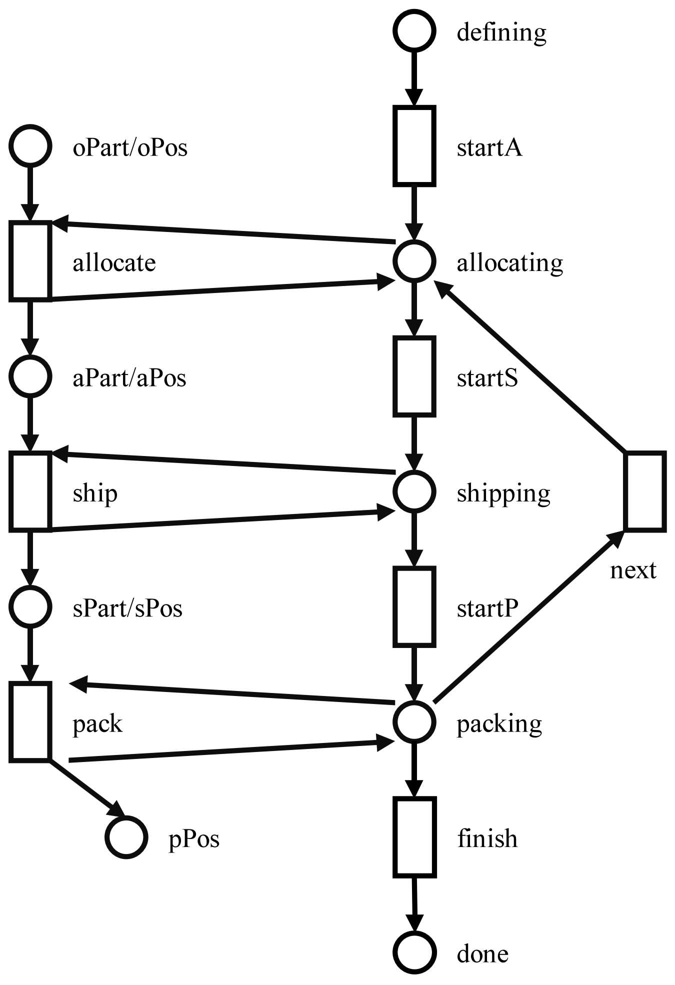


Figure 46. The analysis by the neighbourhood algorithm.

4.5 Integrating Domain Heuristics

This section applies domain heuristics, namely relationship cardinalities, to improve the basic algorithm developed in the last sections. Relationship cardinalities are a very old software engineering concept already contained in the relational database and the entity relationship model [Che77], [Rum91] salvaged them into the Object Modelling Technique from where

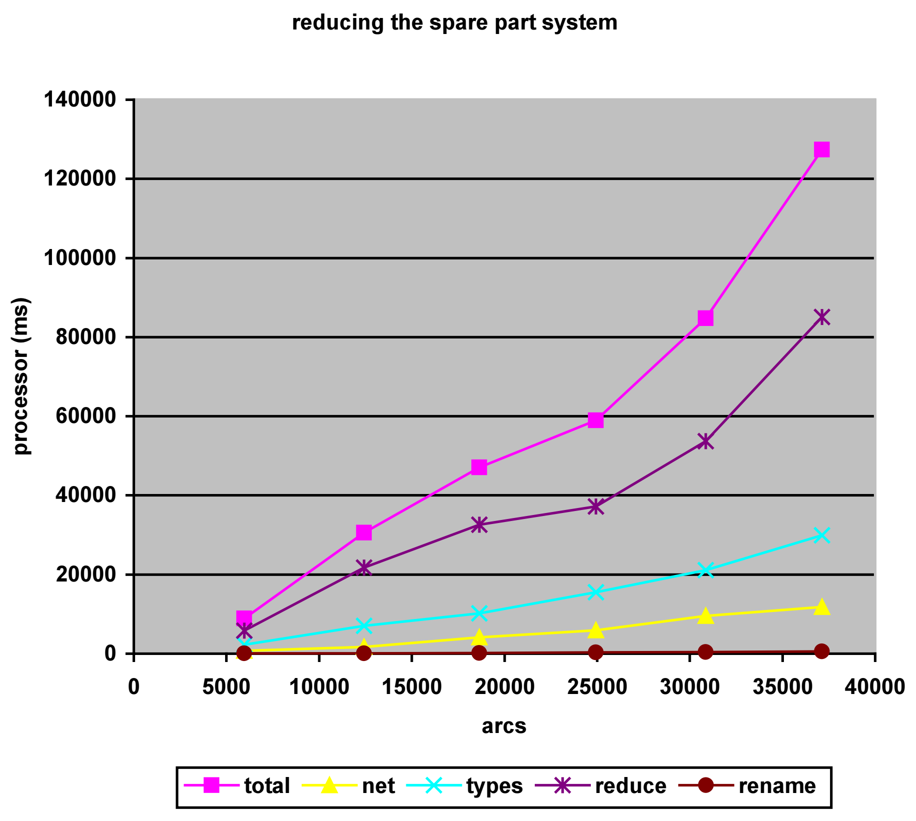


Figure 47. The performance of the prototype.

they found their way into the Unified Modelling Language [rat97]. Also in the following, they will prove to be a powerful concept.

The first task is to separate parts and positions. They may become swapped by subnet automorphisms but they have a 1:n relationship. The trick is to use this asymmetry as an input to the reduction algorithm. We pinpointed in section 3.5.4 that the algorithm offers three dimensions of variations namely similarity, adjacency and choice. This section uses domain heuristics as the similarity criterion whereas the last section varied the adjacency criterion.

After mastering some tricky details, e.g. with non-determinism, such a relationship analysis, yields exactly the reduction net N^u from Figure 29. Hence, for the current example the design is perfectly recovered. But this method not only applies to this specific situation: 1:n relationships are fundamental construct in software engineering, thus it is important to recover them.

The recovered reduction diagram is an important help for human understanding of a system. But it is a reduction which removes information from the original system. Colouring is a Petri net technique to use a small net together with inscriptions to completely specify a large (even infinite) net. So far we practised reverse engineering as discovering reductions. Now we turn to the challenging task of computing a concentration: a concise but complete specification of the original system.

The nodes of a coloured reduction net are inscribed with colour-sets. The colours of the colour-sets correspond to the origins of a node. Usually, colour-sets are built as cartesian products (of elementary colour-sets). But here we will restrict it to an equivalent construction of relationships between colour-sets. The translation between these two concepts is not difficult. This approach simplifies the concept because it does not require a formalism for expressions such as a programming language or a universal algebra. In other terms, yet again, we are able to restrict ourselves to the net-theoretical essence.

Once a reduction is obtained, each reduced node gives a colour-set, namely its origins. But this is rather trivial. The challenge is to detect a helpful structure between these colour-sets. Colour-sets in 1:1 relationships get identified transitively. Relationships between colour-sets that equal or contain each other are classified as the same relationship. Such relationships correspond to structures of the application domain, e.g. the data model.

The enhancement of the reduction algorithm with relationship analyses yields convincing results on our running example:

- The design of the system is completely recovered. This signifies to bridge the huge gap from a low-level net to high-level design information. We recover from a given code the basic concepts of the application domain, namely the fundamental data areas and their relationships.
- However we obtain diagrams that contain more information than our design diagrams. The use of coloured Petri nets leads naturally to novel diagrams for both forward and reverse engineering. In section 4.6.6 we will combine these ideas with concepts from category theory.

4.5.1 Reduction Refinement

To compute finer reductions the reduction algorithm needs the relationships as an input. The type system is available in this early stage. An elementary type path is a tuple $w = (c_b, c_e)$ of two trace classes of the same type. To such a path we define a relationship rel(w) on the places of the source net by

$$\mathrm{rel}(w) = \{(p_b, p_e) \in PxP \mid p_b \neq p_e,\ p_b \in \varepsilon\, c_b \text{ and } p_e \in \varepsilon\, c_e \text{ for an appropriate } \varepsilon \in \mathrm{embeddings}(\mathrm{type}(c_b))\}$$

The relation rel(w) relates beginnings and endings of all paths in the Net that

- lay completely in the image of an embedding ε of the given type
- are not cyclic
- begin and end in the images of the given trace classes

Algorithm 88. Relationships from Types. ⇨144◈

computes all relationships and stores them in a table in `NetStructure`. The table maps paths to relations. This is a slight generalisation over tuples of trace classes that will be needed later.

Relationships in computer science are built over two sets which are defined in advance. This is not the case here. We start from the relationship and thereafter only determine which sets sit at both ends. They should possibly be designated as virtual relationships to indicate the deviation from the usual situation.

At the start of this paragraph we called the used type paths elementary. The co-notated generalisations comes in two different forms

- for the same type system, compose the paths and the corresponding relations.
- compose the types as in section 4.3.2. Then, use the elementary paths of the expanded type system.

These two composition techniques are similar. They share the problem of combinatorial explosions. The first approach composes relationships of path concatenations, the second combines embeddings of types. We will discuss the first technique in more detail later in paragraph 4.5.2.

Looking back to section 4.3.3 the reduction by neighbourhood algorithm generated the equivalence relation from the types. They could be derived from the relationships instead:

Algorithm 89. Membership from Relationships.

```
NetStructure>>membershipFromRelationships
   | mark mShip |
   mShip := Dictionary new.

   self relationships do: [:rel |
        mark := Array with: rel with: #origin.
        rel origin do: [:ori |
           (mShip at: ori ifAbsentPut: [self newSet]) add: mark].
        rel isSymmetric ifFalse: [
           mark := Array with: rel with: #image.
           rel image do: [:ima |
              (mShip at: ima ifAbsentPut: [self newSet]) add: mark]]].
^ mShip   ♦
```

This computes exactly the same result as paragraph 4.4.2. Only a deviation over relationships was taken. But the relationships allow further analyses which may be used afterwards.

How to compute the 1:n relationships? If the nodes on both sides of a 1:n relationship are contained in the same trace class the reduction algorithm cannot separate the two sides. The nodes of both sides are mixed up because their images are mapped to the same node of the reduction. This happened in our running example for positions and parts. So the algorithm should

- detect 1:m relationships within a single trace class
- separate the one from the many side
- classify the nodes accordingly in the membership dictionary)

The method computes two collections:

- `pairs:` a pair consists of two elements of the relation that are related exclusively to each other. A pair is completely symmetric. It needs to be handled in a special way otherwise its symmetry may destroy the general asymmetry of the two sides.
- `oneSide:` this is the collection of elements that are related to more than one element. If the relation is indeed 1:n all these elements must reside on the one-side. In addition, all

elements that they are related to must reside on the many-side. The many side is computed only temporarily in order to verify the consistency of the computed split.

Algorithm 90. Analyse sidedness.

```
Relationship>>analyseSidedness
  | many |
  self assert: [self isSymmetric].
  many := self newSet.
  self
    pairs: Dictionary new;
    oneSide: self newSet.

  self originImagesAllDo: [:ori :imaColl |
    self assert: [imaColl notEmpty].
    (imaColl includes: ori) ifTrue:
      [^ self oneSide: nil; pairs: nil]. "it is not 1:n !"
    imaColl size = 1
      ifTrue: [ (self allAt: imaColl single) size = 1
        ifTrue: [(self pairs includesKey: imaColl single)
          ifFalse: [self pairs at: ori put: imaColl single]]
      ifFalse: [
        self oneSide add: ori.
        many addAll: imaColl]].

  (self oneSide includesAny: many)
    ifTrue: [self oneSide: nil; pairs: nil]. "it is not 1:n !"
```

This method uses heavily the symmetry of the relation which is the receiver. For example, if an element is related to a single element which is in turn related to a single element these two elements form a pair. A test of equality of the two elements is superfluous. The test at the end is essential: if the one side and the many side overlap the relation does not split in a one and a many side. The same is true if an element is related to itself ♦

It is straightforward to add this analysis to the classification:

Algorithm 91. Memberships compatible with sidedness. ⇨145◈

Only a selection of the 1:n relationships are used for the classification. This prevents that overlapping relationships are compromised by the arbitrary distributions of pairs. The non-overlapping 1:n relationships are selected arbitrarily. We did not experiment with domain heuristics because their was no need for it in any of our examples.

In summary, the reduction algorithm using relationship analysis reduces a net N in the following steps

- compute a type system for N
- translate the types into a set of relationships
- split the symmetric relationships into a one and a many side if possible
- classify the nodes of N by the relationships, and refine the classification by a selection of compatible relationship splits

- apply the reduction with neighbourhood algorithm to N with this classification

Applied to the spare part system this yields exactly the design from Figure 29. Hence the design diagram has been fully recovered from low-level data. Of course an invisible difference is left, parts or positions in 1:1 pairs might be swapped but the algorithm has absolutely no information to prevent this.

4.5.2 Colouring

The next step is to retrieve the colouring information. Colouring means assigning colour-sets to nodes and colour relationships to arcs. This allows filling a simple reduction diagram with the full information of the origin net. Moreover by just looking at the colours of the reduction, we get additional insights of the structure of the analysed net.

To recover the colouring the concept of relationships is used again - this time in the concrete instead of the virtual meaning. The following is a step by step explanation of the colouring algorithm.

The first step is to compute the relationships from the reduction. The relationships are the origins of the arcs:

Algorithm 92. Relationships from reduction.

```
NetReduction>>createRelationships

  self structure relationships: Dictionary new.
  self source arcs do: [:arc |
    self structure
      relationshipOrigin: arc from
      image: arc to
      path: (Array with: (self at: arc from) with: (self at: arc to))] ♦
```

A second non-obligatory step is to compose the relationships in order to get relationships over longer paths in the reduction. As previously mentioned this may lead to combinatorial explosions. Hence the following algorithm must be applied with care:

Algorithm 93. Composing paths. ⇨145◈

The third step is to analyse each relationship for itself for relationship cardinality.

The fourth step is to select the colours. More precisely, we have to partition the nodes in the reduction in classes of nodes with the same colour. We do not use the cardinalities of the origin sets for this purpose. We prefer something less arbitrary which relies on stable facts from the analysis. We have chosen the 1:1 relationships. This requires that the arcs of the net generate a bijection rather than the mere existence of a bijection. The following algorithm computes the minimal equivalence relation over the nodes of the reduction that relates two nodes iff they are connected by a 1:1 relationship:

Algorithm 94. Colour equivalence. ⇨146◈

Each equivalence class stands for one colour that colours the elements of the class.

The fifth step creates the colour-sets and assigns the colours to the nodes of the source nets:

Algorithm 95. Colouring the nodes. ⇨146◈

The colouring is done compatible. I.e. the arcs corresponding to the relationships in connections are coloured with the identity relationship over the colour-set.

The sixth step is the last: colour the arcs of the reduction net. But this is straightforward. For every arc in the reduction we already computed a relationship. We translate this relationship on source nodes to relationship on colour-sets and eliminate duplicates. So we get a set of colour relationships.

If we apply this six-step algorithm to the spare part system and pack the information in a

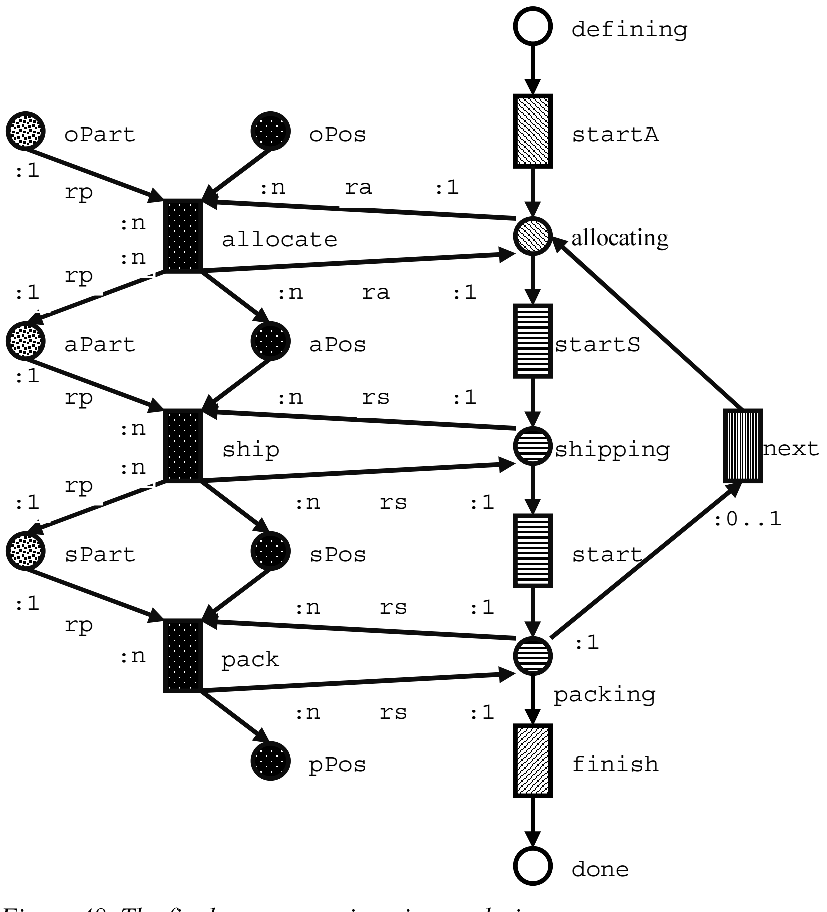


Figure 48. The final reverse-engineering analysis.

single diagram we get Figure 48. The colour-set of a node is painted as a pattern inside the node. There are six colour-sets besides the neutral one for the places `defining` and `done`. There is so much information for arc colouring, that the figure is overloaded although inscription for arcs are dropped iff its colour-relationship is the identity or is used only once

The interesting colour relationships are

- `rp`: which describes the five arcs from parts to the allocate, ship and pack transitions

- `ra`: colours the two arcs between the transition `allocate` and the place `allocating`.
- `rs`: colours the four arcs between the transitions `ship` and `pack` and the place on their right hand side.

The `startA` and `finish` transitions have different colours, although, in reality, they are 1:1. The problem is that the way between them is too long, especially because the branch over next, that seems shortest is a dead end: a `packing` place in the origin net is either connected to `next` or to `finish`, never to both. But we can identify these colours, just by composing colour relationships to paths of the necessary size. This modification does not have any bad side effects it only identifies the two colours wanted. This is a sign for the stability of the algorithm.

In fact this diagram is better than our analysis diagram from section 4.2.1 Of course it was information that we read in between the lines when implementing the system. This illustrates the fact, that coloured Petri net notation is potentially more precise than the usual software engineering notation. And the presented algorithm is able of recovering this additional information.

Another interesting fact is that our algorithm finds the net reduction without search. This means that we do not have to bother about evaluating alternatives. We can select them in advance by tailoring the algorithm to our need.

4.6 Extensions, Variations and Applications

The last chapter culminated in a reverse-engineering algorithm able to discover better analysis diagrams for the spare-part system than the ones we have started from. Hence, the major goal of this work has been reached. This final chapter highlights relationships to other approaches, variations and possible applications.

We mentioned several times that the algorithm offers three dimensions of variation. However, up till now, only adjacency and similarity have been elaborated. Choice will be explored in the next section.

There are different strategies to deal with choices:
- Up to now, all the possibilities have been applied together. This approach is in line with universal constructions, leading to a unique solution and avoids combinatorial explosions.
- If relevant information is available a rational choice becomes possible. Examples are the feedback mechanism from section 4.3.4 and the domain heuristics in section 4.5.
- An simple method is selecting an arbitrary alternative. As depicted in Figure 17 this is an appropriate strategy in situations that require a pure alternative. Complications arise if there are dependencies between multiple alternatives. E.g. in the current example a merger of `aPlace` and `aPosition` is only avoided if all choices for all origins are compatible.
- Avoid the choice. Instead, analyse at the structure of the set of all alternatives.

A motivation for the last strategy is that at the time when a choice is selected the consequences are not yet known. In the end, certain choices will yield the same result and many choices will not make sense. But this may be revealed by the analysis of the complete solution set and enable rational choice. A formalism based on relations is introduced. It allows to represent and compute the set of all different choices in product form.

Reductions are morphism from the net under investigation whereas components and patterns are morphisms from a simpler net into the analysed net. Typically these morphisms are epimorphic for reductions, monomorphic for components and more general for patterns.

Components are building blocks of a system. They should have a small interface, high cohesion, low adhesion and be reusable. Traditionally components are detected as follows:

- The system is split up into subsystems, e.g. by syntactical means.
- The similarity of the subsystems is measured and similar ones identified with reused components

Contrarily, our approach first detects similar structures and then interprets them as components. Its capability for component detection is tested on a net with a deeply nested structure. Although this method has some restrictions it detects interesting components.

Next, the integration of a cliché library in the developed method is studied. In general such a library is not necessary for us but there is a natural point for integration, which allows co-operation between the two approaches.

The experience of many years of reverse engineering demonstrates that no single analysis works for all cases. There are too many different styles of implementation and too many ideas of design. So we have to use a wide variety of algorithms to analyse and combine different analysis results. In reverse engineering that it is often better trying different light-weight methods than spending excessive resources on a single analysis. The results from such a deep search are often trivial or unusable. From this observation some ideas of how to integrate our method in a reverse-engineering framework are derived.

Next, a multi-level application of the algorithm is presented. If a first reduction is not satisfactory then a second one with a different merging criterion is applied. In the current example this allows to deal with less homogeneous input.

In this work many Petri nets are used to explain software systems. They seem a powerful model for systems even ignoring concurrency. Similarly, folding-based techniques seem valuable in areas traditionally occupied by purely clustering-based methods. In conclusion, we advocate Petri nets as an engineering metaphor. As a visualisation we introduce commutative net diagrams. These combine commutative diagrams from category theory and Petri nets and highlight properties of relationships. These properties are important for system understanding, implementation and tuning but normally not made explicit.

4.6.1 Choice

The goal of this section is to compute the set of all possible choice reductions of a net. To avoid combinatorial explosions during both the computation and the analysis, a product form of the solution set is indispensable. Section 4.4.1 described the central data structure of the reduction algorithm as an equivalence relation which reflects the mergers of nodes during the algorithm. Here, the (intermediate) sets of solutions are expressions in relational algebra. We will remain on this abstract level and refrain from details. In particular, no attempt is made to determine a cost bound.

Definition 96. The following operators work on relations:

- Id_P = the identity relation on P (P may be omitted)
- $\pi \subseteq \pi'$: π is finer then π', or π' is coarser, which means that the tuples of π are a subset of those of π'
- $\pi' \cap \pi''$ = intersection of the two relations
- $\pi' \cup^* \pi''$ = minimal equivalence relation containing the tuples of both relations
- $\pi|\sigma = \{(r, s) \in \pi \mid r \in \sigma \text{ and } s \in \sigma\}$ the restriction of a relation to a subset of its base set.

For sets of relations Π' and Π'' define:
- $\Pi' \leq \Pi''$ iff ($\forall \pi' \in \Pi' \; \exists \pi'' \in \Pi''$ with $\pi' \subseteq \pi''$) and ($\forall \pi'' \in \Pi'' \; \exists \pi' \in \Pi'$ with $\pi' \subseteq \pi''$)
- $\Pi' \text{ op}^x \Pi'' = \{\pi' \text{ op } \pi'' \mid \pi' \in \Pi' \text{ and } \pi'' \in \Pi''\}$ for any binary operator op on relations.
- $\underset{\pi\in\Pi}{\text{op}^x}(\pi)$ = if $\Pi = \{\}$ then Id_P else $\pi' \text{ op}^x \left(\underset{\substack{\pi\in\Pi \\ \pi\neq\pi'}}{\text{op}^x}(\pi) \right)$ for $\pi' \in \Pi$

Let N be a net and select one type of reduction, e.g. choice reductions with fixpoints. Let Φ the set of relations corresponding to the set of all such reductions:
- the elements of Φ are equivalence relations over the nodes of N
- Φ contains coarsest relations (i.e. $\psi \in \Phi$ with if ϕ, $\psi \in \Phi$ with $\phi \subseteq \psi$ then $\phi = \psi$)

The first thing to do with Φ is to look at the union and the intersection:
- $\Phi^\cup = \cup^{*x} \Phi$
- $\Phi^\cap = \cap^x \Phi$

We have already calculated $\Phi^\cup$ by the computation in section 4.4.1 using the strategy to apply all choices together.

$\Phi^\cap$ does not merge parts and positions as shown in Figure 49. It is not even a net because the `allocate`, `ship` and `pack` transitions are merged but their surrounding places are not. However, a comparision of Figure 49 for $\Phi^\cap$ and Figure 46 for $\Phi^\cup$ clearly reveals that further investigations on parts and positions are needed whereas the rest of the net is already understood. Furthermore, a local analysis becomes possible, e.g. on `aParts` and `aPos`. Thus it may profit from better performance and separation of concerns. In the current example the choices in different clusters are independent. This yields the following product structure of Φ:

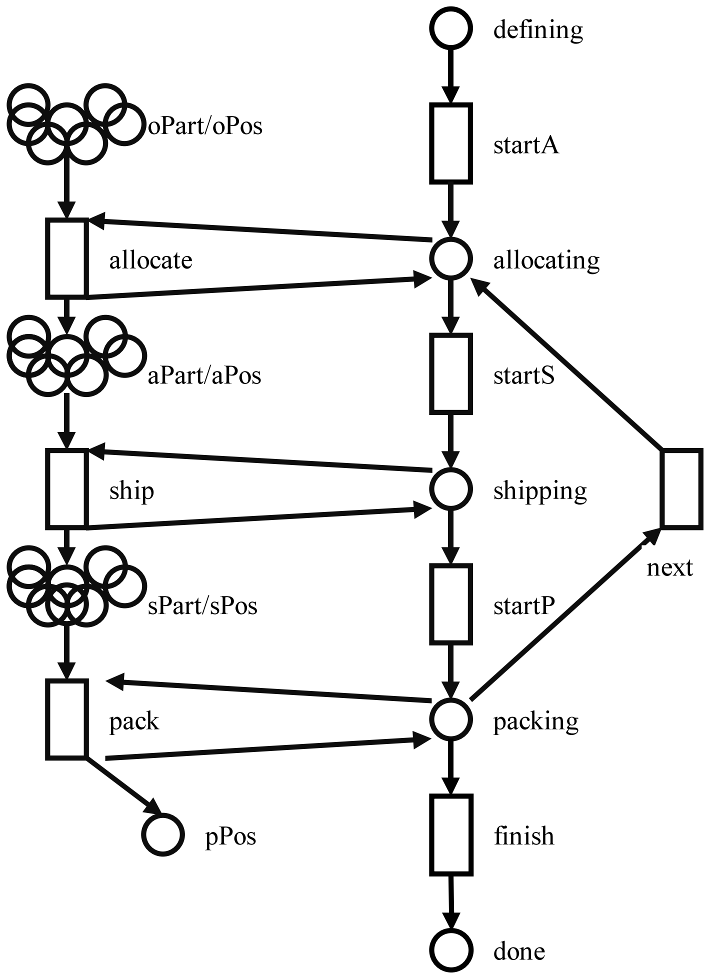


Figure 49. The intersection $\Phi^\cap$ of all reductions.

- Let $r^{\cup} : N \rightarrow N^{\cup}$ the reduction from N to the net $N^{\cup}$ corresponding to the relation $\Phi^{\cup}$.
- $\Phi_X = \bigcup_{\varphi \in \varphi} \varphi \,|\, r^{\cup^{-1}}(x)$ the projection of Φ to the origin of each node x in the reduction $N^{\cup}$
- $\Phi = \bigcup_{x \in X^{\cup}} {}^{*x} \Phi_X$

I.e, the relation set Φ is union of the projections to the origins of the nodes of the reduction $F^{\cup}$. This factorisation is used as an inspiration for the general case.

A tree will represent a set of relations. Each node n of the tree is assigned a relation $\rho(n)$. A subtree with root n represents a set of relations $\Pi(n)$

$\Pi(n) = \{\rho(n)\} \bigcup^{*x} \{\Pi(c) \,|\, c \text{ a child of } n\}$

$\omega(n) =$ if n is the root then Id else if p the parent of n then $\omega(p) \cup^{*} \rho(p)$

The relation $\omega(n)$ is combined from the path from n upwards to the root and contains everything that is unique for n whereas the children deal with the variations. For any set C that contains a node of every path from each leaf to the root r of the tree holds

$$\Pi(r) = \bigcup_{n \in C} \{\omega(n)\} \cup^{*x} \Pi(n)$$

Algorithm 97. Computes all reductions using relational algebra. ⇨147◈.

The algorithm starts with a tree which consists only of a root and the identity relation. Step by step it is mutated into a tree representation of Φ. First the unique parts are processed and only thereafter the variations downwards in the tree.

4.6.2 Component Detection

Because the current example does not have a component structure the telephone net from Figure 37 will be used to test the component detection capability of the reduction algorithm. This net comes from a different application area from a different software engineering method. Thus it is really a hard test for our method. The component structure is deeply nested:

```
telefon: 10 arcs
  hook : 1 transition , 1 place 2 arcs
  dialer: 2 transitions. 1 place, 13 arcs
    tonePulse: 1 place 1 arc
    tone_dialer: 3 transitions, 3 places, 14 arcc
      dtmf => 7arcs
        {HighFreq. LowFreq}: 3 transitions, 5 Places, 15 arcs
        a: 1 transitions, 2 places, 5 arcs
    pulse_dialer: 5 transitions, 4 places, 14 arcs
  keyboard: 1 transition, 1 place, 2 arcs
  controller: 6 transitions, 9 places, 35 arcs
    stateMachine: 5 transitions, 5 places, 25 arcs
      dialState: 7 transitions, 10 places, 79 arcs
        {h, h1, h2, h3, h4, h5}: 2 transitions, 3 places, 8 arcs
        {dial, redial, onHook}: 7 transitions, 10 places, 27 arcs
  line: 8 transitions, 5 places 16 arcs
```

Some of the components consist of more interfaces than internals, e.g. those without nodes but with many arcs. Running the neighbourhood reduction over this (unfolded) net yields 6 colour-sets:

- a set of 3 colours that colours 15 of the 17 nodes of {dial, redial, onHook}
- a set of 2 colours that colours the remaining 2 nodes of {dial, redial}. These two nodes are an input place and an input transition.
- a set of 6 colours containing 3 of the 5 nodes of {h, h1, h2, h3, h4, h5}
- a set of 4 colours with the two remaining nodes of 4 components of{h, h1, h2, h3, h4, h5}
- a set of 2 colours with the two remaining nodes for the two remaining components of {h, h1, h2, h3, h4, h5}
- a 1 colour-set, for all the remaining nodes.

This result at first sight looks rather disappointing: we detected only two of fifteen components and even these two components are different.

But 12 of the 13 components not detected are used only once. Our reduction and colouring method has no chance to detect a single occurrence of a component. Anyway, it might not be a bad idea to focus on components actually reused rather than on components only claimed to be reusable. With this principle only one out of three components are lost. The two instances of the lost component are connected in a way that conflicts with the neighbourhood definition used. This is another principle: a specific adjacency criterion is applied.

The differences between the detected components and the design are border effects. It is rather surprising that they are so small. Principally they cannot be avoided but only minimised by tuning the analysis.

In conclusion, the neighbourhood reduction algorithm uses two principles, namely
- it detects components actually reused (used twice or more)
- if they are connected compatible with the adjacency definition.

Under these preconditions and although the reduction algorithm was not developed for this task it detects interesting components.

4.6.3 Using a Cliché Library

A cliché Library may be used to explain a program in terms of a set of basic programming clichés from the library. Whether this will scale to a real-word programs is still subject to active research. The approach is especially attractive within the current trend of thinking in patterns [Gam95].

The developed reduction algorithm does not need a cliché library. This has the advantage, that the algorithm finds structures actually reused and such a library needs neither to be constructed nor to be searched. The latter is expensive – NP-hard (in the size of the library not in size of program [Woo98]).On the other hand, structures that are used only once cannot be discovered, even if they are very common.

The challenge of cliché recognition is that a single cliché can be applied in many ways. Take the example of a controlled loop. It can be so simple that it fits on half of a line, for instance the following print-out of the square below 1000:

```
1 to: 1000 sqrt do: [:i | i * i printOn: aStream cr]
```

Such a loop can be decomposed in six parts
- a start expression for the index
- a stop expression
- an increment expression
- a comparison of the index with the stop expression
- the block of code executed for every index
- the increment of the index

and each of these parts can grow to arbitrary size and complexity. Also, a programmer has the choice between many constructs offered by the language or the class library. Thus, a cliché may show up with unexpected deformations – and it is in these cases that reverse engineering is most needed. Furthermore, it is not very useful to extract all instances of all clichés. Crucial is to select the clichés that explain the pertinent piece of code.

A natural way to integrate a cliché library in the reduction algorithm is to insert the clichés in the type system. Section 4.3.1 described how to build the type system from the transitions in the investigated net and section 4.3.2 how to compose them. But computing costs become exorbitant for large compositions. However, if the type system is initialised from a type library using only a single composition path the overhead is reduced. Furthermore, there is a hope that there are meaningful clichés that are easier to recognise.

Using the type system to detect clichés, means detecting subnets or monomorphisms. This is far away from the flexibility needed to recognise clichés as explained above. To compensate for this inflexible matching we propose to run the matching in different stages of the reduction. At first, it seems that such a technique has an extremely low probability of detecting a cliché. However, at a speculative level, there are two arguments as to why the chance would not be so bad:

- A reduction algorithm concentrates certain aspects of a program text. If it finishes, it should contain a certain abstraction rather than a random transformation. If the cliché library contains the typical results the chances for matches get improved.
- Inserting the cliché library into the type system implicitly modifies the search strategy. A possible improvement is to explicitly modify the search strategy so that the search paths get attracted by the clichés nearby.

4.6.4 Flat Search

During the research for this work, we experienced the following many times: When the system required a long time for a task then it looped around a programming error or the results were not useful. For example composing large types is very slow and the resulting reduction gets split up in too many different cases. This phenomenon shows a principle that is true for many methods in reverse engineering:

- It is better to use several flat searches than a single very deep search.

Its nearly linear cost definitely makes the developed algorithm to a candidate for a light-weight analysis in a reverse-engineering framework encompassing various different analyses. This 'take it easy' principle may be formulated as a funny application of complexity theory:

- Here is a hard problem and a set of algorithms. Solve the problem by applying an algorithm as long as it runs fast. As soon as it starts to slow down switch to another algorithm.

This would translate to a reverse-engineering framework as follows:

```
[self isProblemSatisfactoryAnalysed] whileFalse: [
  self tryToAnalyseFurtherBy: self chooseAnAnalysisAlgorithm]
```

Syntactically this is an algorithm but it contains mainly open questions:
- how to measure whether an analysis is satisfactory
- how to process results of one algorithm by another
- how to decide whether an algorithm is worth trying
- how to detect when an algorithm passes from useful work to the complexity dead end.

4.6.5 Reducing the Reduction

For the current example the reduction algorithm yields a perfect analysis, but it is still possible to make it more difficult. The data we analysed up till now, always used complete orders. I.e. each order started with a token in `defining` and terminates with a token in `done`. But a system trace for an arbitrary time interval contains orders already started at the beginning and not finished at the end of the interval.

To investigate the related problems the input data is modified to contain orders stopping
- at `defining`: complete order as used till now
- at `packing`: order still has to do more `pack`s or to `finish`
- at `allocating` after a `next` transition: order has still more work.

For such an input the algorithm computes the reduction shown in Figure 50. The `packing` place becomes doubled. The `packing`

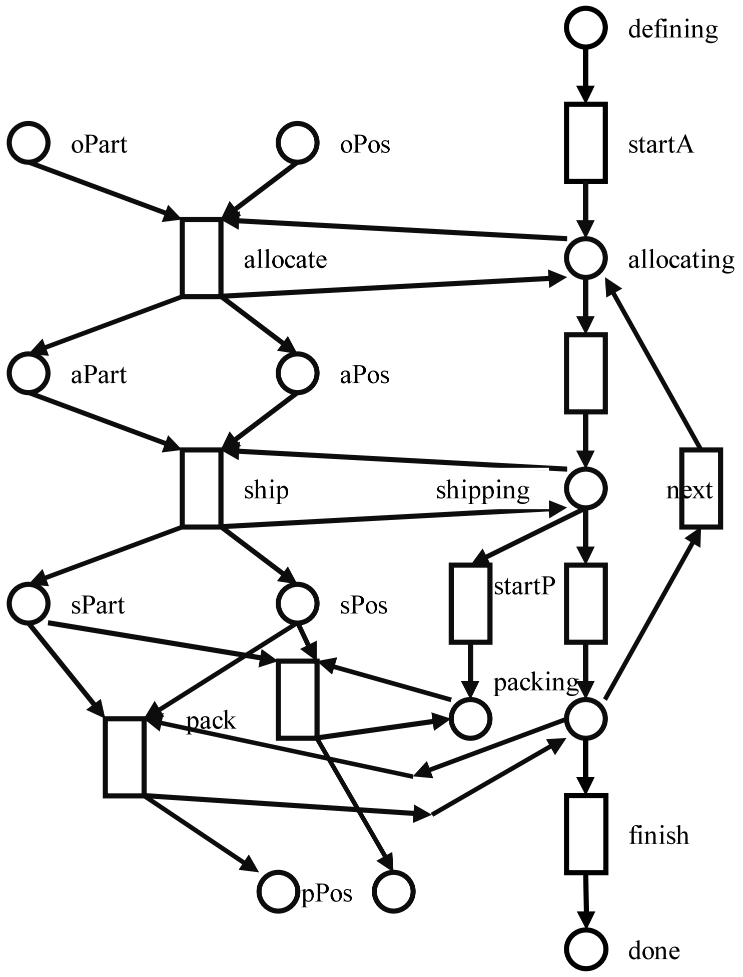


Figure 50. Analysis of incomplete orders.

places in the origin net, that are connected to a `next` or a `finish` transition are separated from the places that are connected to none of them. This split propagates to the `startP` and `pack` transitions and to the `pPos` places. Furthermore the places get additional colours.

To get a further reduction from this net the neighbourhood reduction is applied a second time with a different merger criterion. Node and their colour-sets will become merged in a compatible way.

The situation for the place `shipping` and the two `startP` transitions should be transformed as depicted in Figure 51. The first step allows merging the two transitions

- either because their colours are already embedded in a bigger colour,
- or because the two colours can be merged.

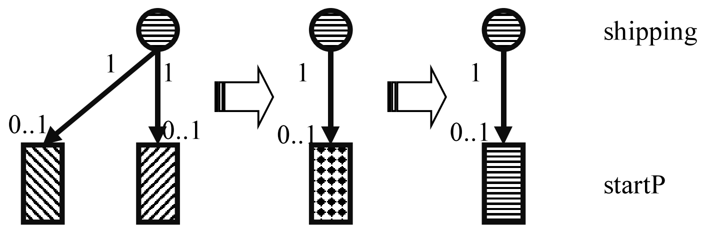


Figure 51. Colour mergers.

The second step detects that the new colour can be embedded in the colour of shipping and identifies them.

What are the necessary checks?

- The second reduction induces a map of the colours.
- The initial origins of a reduction node are 1:1 with its colours. This may be weakened to a subset relationship, if the colours of merged nodes form disjoint subset of the new colour-set.
- The colour mergers should not be arbitrary but reflect facts from the net. This will lead to a more natural modelling with more arcs labelled with identities or functions carrying semantic
  information.

To keep track of colour embeddings we use a class `ColourWood`. As the name suggests it contains a collection of trees. This collection is a graph with colours at the nodes and colour relationships (embeddings) at the arcs.

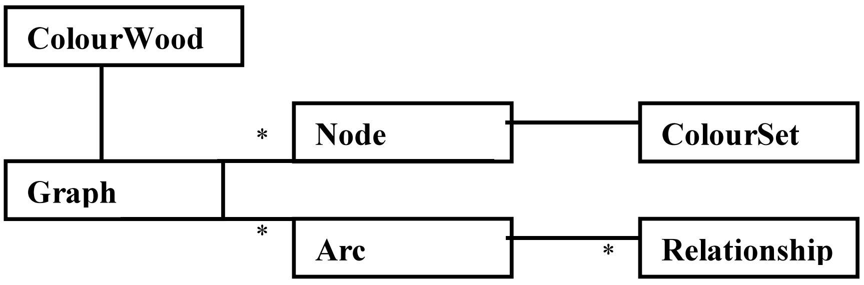


Figure 52. The class diagram for colour reductions.

Up till now, Algorithm 84 used a collection of maps of allowed transition mergers. Now a filter is applied which tries to find a consistent colouring. Moreover the colour structure which must be kept in sync with the transition mergers, this is detailed in

Algorithm 98. Merging colours and nodes. ⇨147◈

Applying this algorithm to net from the beginning of this section gives the complete analysis from Figure 48. Unfortunately this success does not reappear for more complicated situations. If for example orders stop and start at any of the right hand side only, complete reduction is no longer reached. There are two areas the second algorithm needs to be enhanced:

- colour mergers do not only issue from 1:{0,1} relationships, but also from 1:n relationships
- flexibility allowing to use colour subsets.

These modifications are not difficult to implement but another problem arises: the reduction merges things that it should not. To make this second reduction a powerful and stable instrument a careful balancing of the different requirements is necessary, as we have done it for the neighbourhood reduction algorithm. But this is left to further research.

4.6.6 Commutative Net Diagrams

In section 4.5.2 Colouring we got a reverse-engineering diagram that was superior to the usual forward engineering design diagrams. This has two interesting consequences:

- Reverse engineering forces a closer look at the forward engineering methods. One detects information that is essential for the correct functioning of a software system but is not reflected in the forward engineering process. This is an instance of phenomenon in AI: the automation of a human activity often fails if it considers only what humans think they do. Usually the unconscious decisions and actions are crucial. Reverse engineering may help to detect some of these implicit decisions, so they might become explicit or event built into a method.
- Reverse engineering has to recover design. Maybe this is so difficult because important parts of the design process are not explicit. As reverse engineering uses finished software systems it is confronted with intimately interlaced effects of the explicit and the implicit design. Analysis restricted to the explicit parts alone is only applicable in the most simple situations which do not really need reverse engineering.

In this paragraph, we mention such a hidden concept. The diagram in Figure 48 can still be enriched by information about commutative relationships and relationships composition. To our knowledge the concept of commutativity has never been used in the context of software analysis. Nevertheless it plays an important role for semantics, implementation and optimisation.

Commutative diagrams are a corner stone of category theory. Entity Relationship diagrams or Petri nets diagrams can easily be interpreted as diagrams in category theory. But the term commutative gets a bit more complicated in our case. The reason is that relationships run in both directions while morphisms are one way. Often, although a whole mesh might not be commutative there are some important properties of 'local commutativity'.

Such generalised commutativity diagram for our spare part system is shown in Figure 53. Each block arrow drawn concerns the elementary net mesh it resides in. The two types of block arrows show two independent features:

- the circular arrows designate an elementary cell as commutative
- a straight arrow pointing to a node n says that the composition of the relationship around the whole mesh from n to n equals the identity relationship.

- relationship cardinalities are a third, different feature. They are not shown here but they can be read of the colour diagram

Commutative cell - indicated by a pair of circular arrows - signifies the relationships along the two paths around the circuit between two arbitrary nodes compose to the same relationship. As an example take the left upper cell. From `aPart` to `aPos` there are two paths, one over

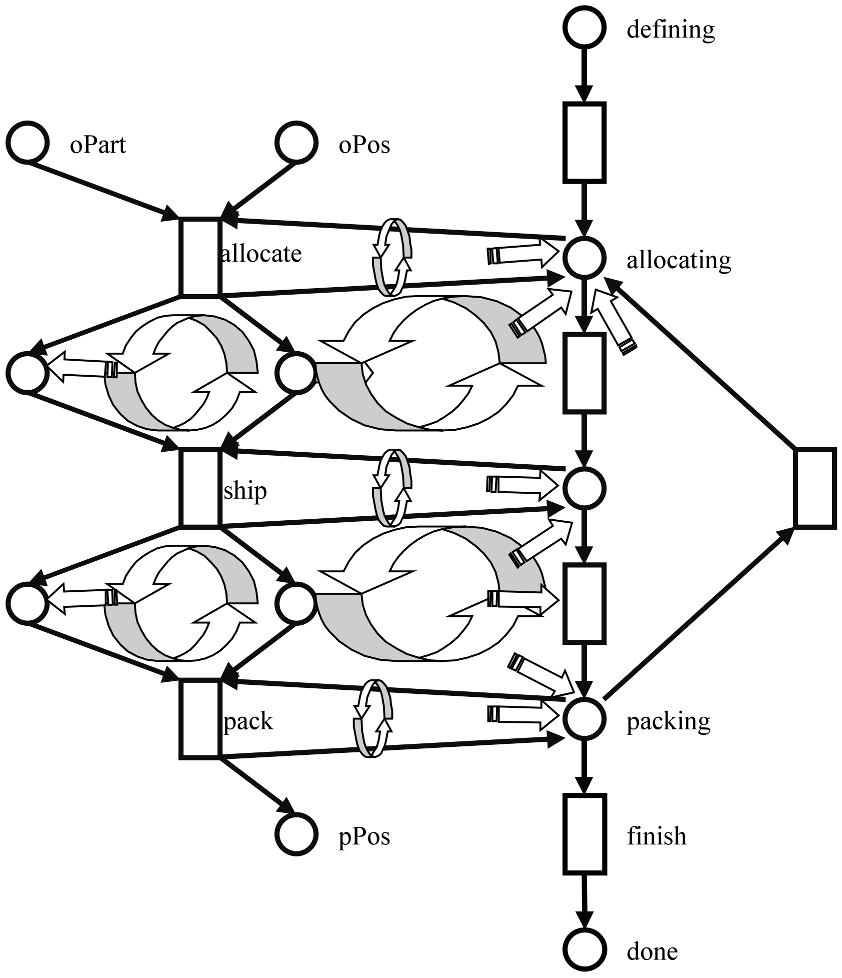


Figure 53. A commutative net diagram.

the transition `allocate`, the other over `ship`. Both compose the same relationship that relates a specific part with all positions containing this part. This is an important semantic information. It expresses that the many arrows form a single relationship from a user point of view. Furthermore this constraint is crucial for understanding, simplifying and tuning object navigation and database accesses.

The place `aPart` is indicated by the straight arrow as an ‘identity point’ in the same left upper cell. This means a run around the loop from a specific part always returning to the same part. This is not true for the transition `allocate`: the run may return to any `allocate` transition of the same part, even belonging to a different order. Again such 'stabilising elements' arc crucial for understanding of the application.

An application of these ideas would be a further reduction step in the reverse engineering algorithms proposed in this paper. It would allow the merge of colours only if they are compatible and reside in a commutative mesh. Again we refrain from elaborating any details.

We showed the commutative decorations in a Petri net diagram because we wanted to reuse our running example. But this technique is not confined to nets. The explanation above suggests that both class or entity-relationship diagrams can get decorated with such symbols for cell commutativity. In fact this notation makes sense in any diagram with edges symbolising a kind of relationship.

Of course there are more complicated situations that this simple notation cannot reflect. But we do not aim for a complete classification of properties of relationships. Rather we want to pinpoint novel analysis techniques for forward or reverse engineering arousing naturally from our approach of using coloured Petri nets as an engineering metaphor.

5 Conclusion

The net-theoretical part of this PhD thesis builds adjunctions between Petri-net categories standing for the dichotomies of clustering and folding as well as of structure and behaviour. They form a bridge between applications of Petri nets using clustering-based techniques for software engineering and folding-based categorical methods offering deep theoretical insights. The hope is that this bridge will allow the different paradigms to benefit from each other. Furthermore, the categorical framework extends to coloured nets and semantics and offers integration with graph-transformation techniques of different flavours.

The application of universal constructions from these categories to reverse engineering yields a new analysis algorithm which can recover high-level design information from a flat net. This is done without searching and hence at high speed - almost linear. At the same time, the algorithm offers a wide range of variations allowing it to be adapted to different application domains. As it is a Petri-net based method, most problems need a preceding translation to a net – which may be done in many different intuitive ways.

In this work, we looked for synergies between Petri nets and reverse engineering and were able to discover some attractive new techniques. With minor adaptations, many conventional clustering-based reverse-engineering tools would apply to Petri nets. In fact they really are necessary there. Nevertheless, we concentrated on the unconventional approach by applying folding-based Petri-net techniques to the reverse engineering of legacy systems. Consequently, the basic character of this work remains experimental and many questions must remain open. We can only select a few:

- We showed how to formalise coloured Petri nets. However, it is more challenging and of higher practical relevance to do this for hierarchical nets. On a theoretical level, it means to complete the cube of Figure 27 with the three axes folding/clustering, structure/behaviour/semantics and low-level/high-level.
- The use of examples from different application domains to validate and improve the reverse-engineering algorithms proposed in this work.
- Integration of the algorithms into a reverse-engineering framework to find heuristics for the situations in which they are preferable to current methods.
- Huge and confusing search spaces arise in reverse engineering. We were able to navigate through them efficiently with the aid of universal constructions. Universal constructions may also prove useful in other ways and in other domains to combat combinatorial explosions.

The dialectic between the static graph of a net and its behaviour is one of the essentials of net theory. Although obvious correspondences exist in software engineering, this work was unable to make use of them. Translating a system to a bipartite graph already offers powerful modelling and analysis features. In terms of intuition, however, the driving force is the Petri-net dynamics behind the static structure. We conclude that folding-based Petri net methods are a powerful tool in both forward and reverse engineering and that it is also worthwhile devoting further research to them in other domains.

6 References

[Aal97] W.M.P van der Aalst. Verification of Workflow Nets. 407-426 in [ATPN97].

[And96] Roland Martin Andersson. Reverse Engineering of Legacy Systems: From Value-Based to Object-Based Models. Thèse No 1521 (1996) EPF Lausanne.

[Ant97] G. Antoniol, R. Fiutem, G.Lutteri, P. Tonella, S.Zanfei, E. Merlo. Program Understanding with the CANTO Environment. 72-81 in [ICSM97].

[APN86] G. Goos, J Hartmanis (Eds.). Petri Nets: Applications and Relationships to Other Models of Concurrency. Advances in Petri Nets 1986. Part II. [LNCS] 255.

[APN93] Grzegorz Rozenberg (Ed). Advances of Petri Nets 1993. [LNCS] 674.

[APNC99] J. Billington, M. Diaz, G. Rozenberg (Eds.). Application of Petri Nets to Communication Networks. 1999. [LNCS] 1605.

[ASI98] Jieh Hsiang, Atsushi Ohori (Eds.). ASIAN'98. 1998. [LNCS] 1538.

[ATP82] Claude Girault, Wolfgang Reisig (Eds.). Selected Papers from the First and Second European Workshop on Application and Theory of Petri Nets. 1982. [IF] 52.

[ATPN93] Marco A. Marsan (Ed). Application and Theory of Petri Nets 1993. 14th. Int. Conference, Chicago. [LNCS] 691.

[ATPN94] Robert Valette (Ed). Application and Theory of Petri Nets 1994. 15. Int. Conference, Zaragoza. [LNCS] 815.

[ATPN97] Pierre Azema, Gianfranco Balbo (eds.) Application and Theory of Petri Nets 1997. [LNCS] 1248.

[ATPN98] Jörg Desel, Manuel Silva (eds.). Application and Theory of Petri Nets 1998, 19th International Conference, Lisbon, Portugal. [LNCS] 1420.

[Avr95] Denis Avrilionis, Pierre-Yves Cunin. Using Views to Maintain Petri-Net-Based Process Models. 318-326 in [ICSM95].

[Bak95]Brenda S. Baker. On Finding Duplication and Near-Duplication in Large Software Systems. 86-95 in [WCRE95].

[Bel97] Bernd Bellay, Harald Gall. A Comparison of four Reverse Engineering Tools. 2-11 in [WCRE97].

[Bou94] Pierre Bourque. An Innovative Software Reengineering Tools Workshop (Market Maturity and Lessons). 30-34 in [SEN] vol. 19, no 3, July 1994.

[Bra97] Mark van den Brand, Alex Sellink, Chris Verhoef. Generation of Components for Software Renovation Factories from Context-free Grammars. 144-153 in [WCRE97].

[Bru98] Roberto Bruni, José Meseguer, Ugo Montanari, Vladimiro Sassone. A Comparison of Petri Net Semantics under the Collective Token Philosophy. 1998. 225-244 in [ASI98].

[Bur96] Elizabeth Burd, Malcolm Munro. Clazien Wezeman. Analysing large COBOL Programs: the Extraction of Reusable Modules. 238-243 in [ICSM96].

[Bur97] Elizabeth Burd, Malcolm Munro. The Implications of Non-functional Requirements for the Reengineering of Legacy Code. 215-223 in [WCRE97].

[Bur98] Elizabeth Burd, Malcolm Munro. Assisting Human Understanding to Aid the Targeting of Necessary Reengineering Work. 2-9 in [WCRE98].

[Cab76] Thomas Mc Cabe. A complexity measure. IEEE Transactions on Software Engineering. Dec, 1976.

[Can93] G. Canfora., A. Cimitile, M. Munro, C.J. Talor. Extracting Abstract Data Types from C Programs: A Case Study. 200-209 in [CSM93].

[Che77] Peter Chen. The Entity Relationship Approach to Logical Database Design. 1977. Q.E.D. Information Science Inc..

[Chr80] Dimitrios Christodoulakis, Matthias Moritz. Net Morphisms and Software Engineering. 1980. 111-117 in [ATP82].

[Chu96] William C. Chu, Hongji Yang, Paul Luker, A Formal Method for Software Maintenance. 206-216 in [ICSM96].

[Cim98] Aniello Cimitile, Ugo De Carlini, Andrea De Lucia. Incremental Migration Strategies: Data Flow Analysis for Wrapping. 59-68 in [WCRE98].

[COCS95] N. Comstock, C. Ellis (Eds.) Conference on Organizational Computing Systems, Auf. 1995. ACM Press.

[Con84] S.D. Brookes, A.W. Roscoe, G. Winskel (Eds). Seminar on Concurrency. Pittsburgh 1984. [LNCS] 197.

[Concur'92] W. R. Cleaveland (Ed.) Concur'92. Third International Conference on Concurrency Theory, 1992. Proceedings. [LNCS] 630.

[CPE94] Günter Haring, Gabriele Kotsis (eds.). Computer Performance Evaluation, Modelling Techniques and Tools, Proceedings of 7. Int. Conference, Vienna, May 1994. [LNCS] 794.

[Cre97] Katja Cremer. Using graph technology for reverse and re-engineering. 157-167 in [ReTIS97].

[CSM93] David Card (ed.). Conference on Software Maintenance. Montréal, September 27-30, 1993. Proceedings.. IEEE Computer Society Press.

[CSMR97] First Euromicro working conference on software maintenance and reengineering. Berlin 1997. IEEE Computer Society Press.

[Des96] Jörg Desel, Agathe Merceron. Vicinity Respecting Homomorphisms for Abstracting System Requirements. 1996. Bericht 337 AIFB Universität Karlsruhe.

[Deu97] Arie van Deursen, Steve Woods, Alex Quilici. Program Plan Recognition For Year 2000 Tools. 124-133 in [WCRE97].

[Deu98] Arie van Deursen, Leon Moonen. Type Inference for COBOL Systems. 220-230 in [WCRE98].

[Dic93] DiCesare et al. Practice of Petri nets in Manufacturing. 1993. Chapman & Hall.

[Ell95] Clarence Ellis, Grzegorz Rozenberg. Dynamic change within workflow systems. 1995. 10-21 in [COCS95].

[Erm97] Claudia Ermel, Maike Gajewsky. Expanding the Use of Structuring: Formal Jusitification for Working on Subnets. 1997. 44-54 in [PNSE97].

[Ess97] Robert Esser. An Object Oriented Petri Net Approach to Embedded System Design. 1997, vdf, Hochschulverlag. an der Eth, TIK-SchriftenReihe nr. 16.

[Ezp98]. J. Ezpeleta, F. Garcia-Vallés, J.M Colom. A Class of Well Structured Petri Nets for Flexible Manufacturing Systems. 64-83 in [ATPN98].

[FASE98] Egidio Astesiano (Ed). Fundamental Approaches to Software Engineering. First International Conference, FASE'98, Lisbon 1998. [LNCS] 1382.

[Feh93] Rainer Fehling. A Concept of Hierarchical Petri Nets with Building Blocks. 148-168 in [APN93].

[Fel95] Miguel Felder, Angelo Gargantini, Angelo Morzenti. A Theory of Implementation and Refinement in Timed Petri Nets. 1995. 127-161 in [TCS] 202 (1998).

[Fer94] A. Ferscha, G. Chiola. Accelerating the Evaluation of Parallel Program Performance Models Using Distributed Simulation. 231-252 in [CPE94].

[Fiu96] R. Fiutem, P. Tonella, G. Antoniol, E.Merlo. A Cliché-based Environment to Support Architectural Reverse Engineering. 319-328 in [ICSM96].

[Fri97] Olaf Fricke. Data Encapsulation and Data Abstraction with Petri Nets - a Graphical Visualization of Modules. In [PNSE97].

[FST95] P.S. Thiagarajan (Ed.) Foundations of Software Technology and Theoretical Computer Science. 15th Conference, Dec 1995 [LNCS] 1026.

[Gam95] Erich Gamma et al. Design Patterns: Elements of Reusable Object-Oriented Software. 1995. Addison-Wesley.

[Gar79] Michael R. Garey, David S. Johnson. Computers and Intractability. A guide to the theory of NP-completeness. Freeman. 1979.

[Gir97] Jean-François Girard, Rainer Koschke, Gerorg Schied. Comparison of Abstract Data Type and Abstract State Encapsulation Detection Techniques for Architectural Understanding. 66-75 in [WCRE97].

[GLT79] H.J. Genrich, K. Lautenbach, P.S. Thiagarajan. Elements of General Net Theory. 1979. 21-163 in [NTA79].

[GP95] R.J. van Glabbeek, G.D. Plotkin. Configuration Structures. 1995. 199-209 in IEEE Symposium on Logic in Computer Science.

[Gut98] Jens Gutstedt. Efficient Union-Find for Planar Graphs and other Sparse Graph Classes. 123-141 in [TCS] 203, 1998.

[Gup97] A. Gupta. Program Understanding Using Program Slivers – An Experience Report. 66-71 in [ICSM97].

[Hil71] P. J. Hilton, U. Stammbach. A Course in Homological Algebra. 1971. Springer.

[ICSM95] Proceedings International Conference on Software Maintenance, Opio (Nice), 1995, IEEE Computer Society Press.

[ICSM96] Proceedings International Conference on Software Maintenance, Monterey, California, 1996, IEEE Computer Society Press.

[ICSM97] Proceedings International Conference on Software Maintenance, Bari, 1997, IEEE.

[IF] W. Brauer (Ed) Informatik-Fachberichte. im Auftrag der Geselllschaft für Informatik. Springer.

[JACM] Journal of the ACM. New York ACM. http://www.acm.org/jacm

[Jen92] Kurt Jensen: Coloured Petri Nets: Basic Concepts. Analysis Methods and Practical Use. Volume 1, 1992, Springer.

[Kaz98] Rich Kazman, Steven G. Woods, S. Jeromy Carrière. Requirements for Integrating Software Architecture and Reengineering Models: CORUM II. 154-163 in [WCRE98].

[Kel95] Rudolf K. Keller, et al.. Environment Support for Business Reengineering: The Macrotec Approach. 31-40 in Software Concepts & Tools 16(1) 1995.

[Kul98] Bernt Kullbach et al.. Program Comprehension in Multi-Language Systems. 135-143 in [WCRE98].

[Knu78] Donald E. Knuth, Arnold Schönhage. The Expected Linearity of a Simple Equivalence Algorithm. 281-315 in [TCS] 6, 1978.

[Kon97] K. Kontogiannis. Evaluation Experiments on the Detection of Programming Patterns using Software Metrics. 44-54 in [WCRE97].

[Lak97] Charles Lakos: On the Abstraction of Coloured Petri Nets. 42-61 in [ATPN97].

[Lan71] Saunders Mac Lane. Categories for the Working Mathematician. 1971. Springer.

[Lee90] Jan van Leeuwen (ed.). Algorithms and Complexity. Volume A. 1990 Elsevier. The MIT Press.

[Lin98] Christoph Lindenmann. Performance Modelling with Deterministic and Stochastic Petri Nets. 1998. Wiley, cop.

[LNCS] Lecture Notes in Computer Science. Springer.

[Luc97] A. De Lucia, G.A. Di Lucca, A.R. Fasolino, P. Guerra, S. Petruzzelli. Migrating Legacy Systems towards Object Oriented Platforms. 122-129 in [ICSM97].

[Mai97] Christoph Maier, Daniel Moldt. Object Coloured Petri Nets – a Formal Technique for Object Oriented Modeling. 1997. 11-19 in [PNSE97].

[Man96] Spiros Mancoridis, Richard C. Holt. Recovering the Structure of Software Systems Using Tube Graph Interconnection Clustering. 23-32 in [ICSM96].

[May96] Jean Mayrand, Claude Leblanc, Ettore M. Merlo. Experiment on the Automatic Detection of Function Clones in a Software System using Metrics. 244-253 in [ICSM96].

[Meh86] Kurt Mehlborn. Datenstrukturen und effiziente Algorithmen. Band 1. Sortieren und Suchen. B. G. Teubner Stuttgart 1986.

[Men97] Nabor C. Mendonca, Jeff Kramer. A Quality Based Analysis of Architecture Recovery Environments. 54-57 in [CSMR97].

[MMS92] José Meseguer, Ugo Montanari, Vladimiro Sassone. On the Semantics of Petri Nets. 1992. 286-301 in [Concur'92].

[MMS94] José Meseguer, Ugo Montanari, Vladimiro Sassone. On the Model of Computation of Place/Transition Petri Nets. 16-38 in [ATPN94].

[Mir94] J. Mirkowski. A Method for Transforming Behavioural Specifications of Digital Systems into Petri Nets. In 39. internationales wissenschaftliches Koloquium, 1994, TU Thüringen.

[NC95] Mogens Nielsen, Allan Cheng. Observing Behaviour Categorically. 1995. 263-278 in [FST95].

[NRT90] Mogens Nielsen, G. Rozenberg, P.S. Thiagara. 1990. Behavioural notions for elementary net systems. 45-57 in Distributed Computing 4, Springer.

[NS98] Mogens Nielsen, Vladimiro Sassone. Petri Nets and Other Models of Concurrency. 587-642 in [Rei98].

[NTA79] Wilfried Brauer (Ed): Net Theory and Applications. Proceedings of the Advanced Course on General Net Theory of Processes and Systems. Hamburg Oct 1979. [LNCS] 84.

[Obe96] Andreas Oberweis. Modellierung und Ausführung von Workflows mit Petri-Netzen. Teubner 1996.

[Pad98] J. Padberg, Mgajewsky, C. Ermel. Rule-Based Refinement of High-Level Nets Preserving Safety Properties. 221-238 in [FASE98].

[Pal97] Srinivas Palthepu, Jim E. Greer, Gordon I. McCalla. Cliché Recognition in Legacy Software: A Scalable, Knowledge-Based Approach. 94-103 in [WCRE97].

[PNSE97] Petri Nets in System Engineering (PNSE'97), Modelling, Verification, and Validation. Uni Hamburg, Informatik, Bericht Nr. 205. 1997.

[Qui97] Alex Quilici, Steven Woods, Yongjun Zhang. New Experiments With a Constraint-based Approach To Program Plan Matching. 114-123 in [WCRE97].

[Rat97] Rational Software Corporation: Unifies Modeling Language, Notation Guide Version 1.0. 1997. http://www.rational.com/uml/index.jsp

[Ree97] Alyson A. Reeves., Judith D. Schlesinger. JACKAL: A Hierachical Approach to Program Understanding. 84-93 in [WCRE97].

[Rei98] Wolfgang Reisig, Grzegorz Rozenberg (eds.). Lectures on Petri Nets I: Basic Models. 1998. [LNCS] 1491.

[ReTIS97] J. Fyörkös, M. Krisper, H.C. Mayr (eds). ReTIS'97 5th International Conference on Re-Technologies for Information Systems. CASSAM – Computer Aided Software Support and Maintenance. SR österreichische Computer Gesellschaft Band 107, 1998.

[Rum91] James Rumbaugh. Object Oriented Modeling and Design. 1991. Prentice-Hall.

[Sas94] Vladimiro Sassone, Mogens Nielsen, Glynn Winskel. Models for Concurrency: Towards a Classification. 297-348 in [TCS] 170. 1996.

[Sel98] Alex Sellink, Chris Verhoef. Native Patterns. 89-103 in [WCRE98].
[SEN] SIGSOFT Software Engineering Notes. An informal newsletter of the ACM SIGSOFT. ACM, New York.
[Sif97] Michael Siff, Thomas Reps. Identifying Modules via Concept Analysis. 170-179 in [ICSM97].
[STACS84] M. Fontet, K. Mehlhorn (Ed). STACS 84. Symposium of Theoretical Aspects of Computer Science. Paris 1984. [LNCS] 166
[Sto97] M.-A.D Storey, K. Wong, H.A. Müller. How do Program Understanding Tools Affect how Programmers Understand Programs. 12-21 in [WCRE97].
[Sub96] Gokul V. Subramaniam, Eric J. Byrne. Deriving an Object Model from Legacy Fortran Code. 3-12 in [ICSM96].
[Tar75] R.E. Tarjan. Efficiency of a good but not linear set union algorithm. 215-225 in [JACM] 22, 1975.
[TCS] Theoretical Computer Science. ELSEVIER.
[Til93] Scott R. Tilley, Hausi A. Müller, Michael J. Whitney, Kenny Wong. Domain-retargetable Reverse Engineering. 142-151 in [CSM93].
[Vog92] Walter Vogler: Modular construction and partial order semantics of Petri nets. 1992. [LNCS] 625.
[Wat96] Arthur H. Watson, Thomas Mc Cabe. Structured Testing: a Testing Methodology Using the Cyclomatic Complexity Metric. NIST special Publication 500-235.
[WCRE95] Linda Wills, Philip Newcomb, Elliot Chikofsky (Eds). Second Working Conference on Reverse Engineering, July 14-16, 1995, Toronto, Ontario,Canada. IEEE
[WCRE97] Ira Baxter, Alex Quilici, Chris Verhoef (eds.). Fourth Working Conference on Reverse Engineering. October 6-8, 1997. IEEE.
[WCRE98] Proceedings Fifth Working Conference on Reverse Engineering, October 12-14, 1998, Honolulu. IEEE.
[Wig97] Theo A. Wiggerts. Using Clustering algorithms in Legacy Systems Remodularization. 33-43 in [WCRE97].
[Wil96] Norman Wilde, Christopher Casey. Early Field Experience with the Software Reconnaissance Technique for Program Comprehension. 312-318 in [ICSM96].
[WF86] T. Winograd, F. Flores. Understanding computers and cognition. 1986. Reading, Mass. Addison-Wesley.
[Win84a] Glynn Winskel. A New Definition of Morphism on Petri Nets. 1984. 140-150 in [STACS84].
[Win84b] Glynn Winskel. Categories of Models for Concurrency. 1984. 246-267 in [Con84].
[Win86] Glynn Winskel. Event Structures. 1986. 325-392 in [APN86].
[Woo98] Steven G. Woods, Alexander E. Quilici, Qiang Yang. Constraint-based Design recovery for software reengineering. Theory and Experiments. 1998. Kluwer.
[Zho98] Meng Chu Zhou. Modeling, Simulation and Control of Flexible Manufacturing Systems. A Petri Net Approach. 1998. World Scientific.
[Zim97] Armin Zimmermann. Modellierung und Bewertung von Fertigungssystemen mit Petri-Netzen. 1976. Dissertation Technische Universität Berlin.

7 Appendix: Proofs and Details for Petri Nets

7.1 Introduction

Remark 1. Category theory in computer science.

This remark explains the motivation for the use of category theory within this work and reviews a few basic definitions. For a general introduction to category theory refer to [Lan71].

Category theory can be thought of as a smallest common denominator of mathematics – there are other branches of mathematics with this flavour, e.g. logic. Using category theory in computer science, therefore, stands for the attempt to reason about computers in a way conforming to this smallest denominator. There are many reasons for that such as:

- to profit from the huge work done by mathematicians over the centuries
- to inherit some of the elegance and specially the compositional features of mathematics
- to choose by abstract arguments the better models with less try and error processes

It seems reasonable to do some crosschecks between computer science concepts and category theory – without denying the necessity to crosscheck with other branches of mathematics or sciences e.g. sociology. It is especially important to discuss conflicting hints coming from different crosschecks. But this definitely is beyond the scope of this work.

A category $\underline{C}$ consists of objects and morphism. A morphism f: $X \rightarrow Y$ goes from a source object X to a destination object Y and is drawn as a directed arc. $\underline{C}[X, Y]$ or simply [X, Y] designates the set of morphisms from X to Y. The main axiom is that morphisms have an associative composition. Beside, each object X must have an identity morphism $Id_X: X \rightarrow X$. One level higher functors are defined. They map objects and morphisms from one category to another one preserving morphism compositions and identities, i.e. F (gf) = (F g) (F f) and F $Id_X = Id_{F\,X}$.

Surprisingly, in the very thin air of this abstraction many theories share non-trivial patterns. Formulating something with categories and morphisms hence allows comparing it to other theories and helps to discover or build patterns which proved useful in other contexts. For our subject the use of category theory means to stress

- relationships between nets and
- relationships between net classes

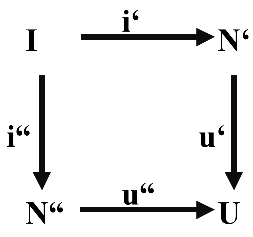
rather than single transitions or places of a net. This is very natural for forward and reverse engineering:

- a net is an implementation of a specification net
- a specification simulates a (projected) implementation
- two components behave similarly and could replace each other

A morphism may model such relationships. Thus, software engineering translates to the construction of a morphism to a given specification and reverse engineering to the construction of a morphism from a running system. Morphism composition corresponds to layered implementation, isomorphism to equivalent implementation and a functor to modelling a formalism by another under preservation of the implementation relationships.

Software engineers like to see a system as static collection of program source text or graphs. But the ultimate goal is the behaviour of the running system. The relationship between structure and behaviour may also be modelled as morphisms or preferably as functors. This modelling implies that composition of structures yields composition of behaviour - a relationship that is far from trivial. Rather it means conciliating conflicting requirements.

Category theory is just a container which poses only minimal requirements on morphisms – the art is to invent morphisms that give the needed flexibility and nevertheless preserve enough properties to allow meaningful analysis.



$$\begin{tikzcd} I \arrow[r, "i'"] \arrow[d, "i''"'] & N' \arrow[d, "u'"] \\ N'' \arrow[r, "u''"] & U \end{tikzcd}$$

Figure 54. A commutative diagram.

The main axiom allows drawing diagrams consisting of objects (e.g. nets) connected by directed arrows which represent morphisms. Figure 54 shows an example. A directed path in a diagram yields the composition of its morphisms. If these compositions equal for every two paths with the same source and destination the diagram is called commutative. Hence in Figure 54 commutative means u'i' = u"i". This is an intuitive, graphical and mathematical sound way to visualise relationships between objects.

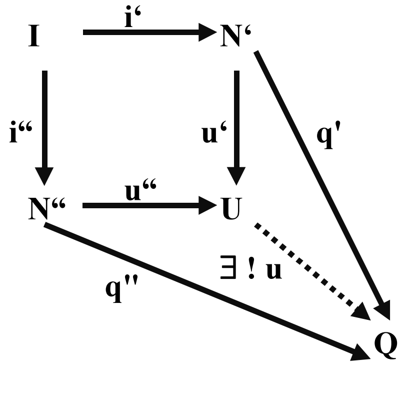
Diagrams allow building new objects and morphisms with especially nice properties – called universal constructions. In fact, there are two types of universal constructions: limits and colimits. The colimit of a diagram D is an object U together with morphisms from every object of D to U having two properties, namely natural and universal.

- Natural means that the combined diagram (of D, U and connecting morphisms to U) is commutative.
- Universal means that for every other such object Q with natural connecting morphisms there is a unique q: U→Q making the whole diagram commutative.

The definition ensures that if a colimit exists it is unique (up to isomorphism).

Figure 54 shows a special colimit (U, u', u") called pushout of the diagram (I, i', i"). Here, natural means again u'i' = u"i" and the universal property is shown in Figure 55. We use ∃! for *there exists one and only one.* In software engineering a pushout is the 'best' combination of two objects N' and N" which are connected by i' and i" to a common interface I. For the computation of a specific colimit and limits refer to the proof of Proposition 3 ⇨104.

Universal constructions arise everywhere in mathematics – even centuries before the invention of category theory in the fifties. A category C̲ is called (co)complete iff every diagram in C̲ has a (co)limit. Checking (co)completeness of a category is a good way to check the soundness of its definition. That does not imply that a category with



$$\begin{tikzcd} I \arrow[r, "i'"] \arrow[d, "i''"'] & N' \arrow[d, "u'"] \arrow[ddr, "q'"] & \\ N'' \arrow[r, "u''"] \arrow[drr, "q''"'] & U \arrow[dr, dotted, "\exists ! u"] & \\ & & Q \end{tikzcd}$$

Figure 55. The universal property.

lacking universal constructions is bad - but one should have a plausible reason for it.

It is a widely used technique to specify languages by grammar rules (e.g. by syntax diagrams or in computational linguistics). Such a rule may simple be defined as a diagram of two morphism l and r with a common source I. This rule directly transforms an object G into an object H iff a diagram such as Figure 56 exists. Figure 56 has two pushout squares. This means that H is produced by replacing L in G by R. This replacement is defined by the rule and the embedding c' of the interface I into the common object C. This simple but flexible procedure is the famous double pushout approach which is used for example in graph rewriting systems.

$$\begin{tikzcd}
L \arrow[d, "g'"'] \arrow[dr, phantom, "\textit{pushout}"] & I \arrow[l, "l"'] \arrow[r, "r"] \arrow[d, "c'"] \arrow[dr, phantom, "\textit{pushout}"] & R \arrow[d, "h'"] \\
G & C \arrow[l, "g"'] \arrow[r, "h"] & H
\end{tikzcd}$$

Figure 56. The rule (l,r) transforms G into H.

This illustrates our motivations to use a categorical approach – but it is used seldom in the Petri net community. In fact there are not too many names publishing in this area. The majority regards the categorical approach as too abstract for real work. Of course this is a general feeling about category theory – but we think that in the field of Petri nets it has also a more specific reason.

Most categorical approaches use morphisms that are pure foldings. This turns out to be very attractive from a categorical point of view – folding morphisms transfer behaviour nicely and allow categorical connections from and to many other models of concurrency opening deep insights. The proposals allowing clustering do not transfer behaviour so nicely – if at all. Hence there is not so much profit from category theory and the results are less exciting.

But in practice clustering is necessary – it is simply indispensable for real-world applications. This is standard software engineering practice not only for Petri nets. A categorical tools which does not support clustering is only of limited practical value.

It is the merit of this thesis to construct new net categories that allow clustering, folding, universal constructions. Hence our work allows using more categorical Petri net theory in software engineering practice. ♦ ⇨23⇦

7.2 Preliminaries

7.2.1 Notation

7.2.2 Sets and Multisets

Proposition 3. SETP is cocomplete and complete.

Proof: For cocompleteness first construct the pushout in SETP. Let Π be a set containing a special element undef and π, π' and π'' be functions with:

- π: S→Π, π': S'→Π, π'': S''→Π are injections
- Π is the disjoint union of π(S), π'(S'), π''(S'') and {undef}

The π's make the embeddings of the given sets in the disjoint union explicit and the special element undef will model elements on which the map must be undefined.

$$\begin{tikzcd}
S \arrow[r, "f'"] \arrow[d, "f''"'] & S' \\
S'' &
\end{tikzcd}$$

For $s \in S$ let

- $\pi\, s\, R'\, y$ iff $\pi'\, f'\, s = y$ or $\pi''\, f''\, s = y$ for $y \in S' \cup S''$
- $\pi\, s\, R'$ undef iff $s \notin \operatorname{def} f' \cup \operatorname{def} f''$
- $R \subseteq \Pi \times \Pi$ the minimal equivalence relation generated by R'
- P the equivalence classes of R not containing undef
- $p: \Pi \rightarrow P$ a partial map defined by $p(s) =$ if ($s\, R$ undef) then undefined else the equivalence class of s in R
- $p' = p\, \pi'$
- $p'' = p\, \pi''$

$$\begin{tikzcd}
S \arrow[r, "f'"] \arrow[d, "f''"'] & S' \arrow[d, "p'"] \\
S'' \arrow[r, "p''"] & P
\end{tikzcd}$$

By construction f', f'', p' and p'' form a commutative diagram in <u>SETP</u> shown in the middle Figure 57. It remains to verify the universal property. Given q', q'' and Q in the lowest diagram spelling out commutativity yields:

- $p'(s') \in \operatorname{def}(q)$ iff $s' \in \operatorname{def}(q')$ and analogously $p''(s'') \in \operatorname{def}(q)$ iff $s'' \in \operatorname{def}(q'')$
- if $s' \in \operatorname{def}(q')$ then $q(p'(s')) = q'(s')$ and analogously if $s'' \in \operatorname{def}(q'')$ then $q(p''(s'')) = q''(s'')$

$$\begin{tikzcd}
S \arrow[r, "f'"] \arrow[d, "f''"'] & S' \arrow[d, "p'"] \arrow[rdd, "q'"] & \\
S'' \arrow[r, "p''"] \arrow[rrd, "q''"'] & P \arrow[rd, dotted, "\exists ! q"] & \\
 & & Q
\end{tikzcd}$$

Figure 57. Pushout: starting point (top), pushout square (middle) and universal property (bottom).

This completely determines q as the images of p' and p'' cover P. But it remains to show that q is well defined. Otherwise there were $p'(s') = p''(s'')$ with $q'(s') \neq q''(s'')$. Let

- $X = \pi'\, p'^{-1}\, p'\, s' \cup \pi''\, p''^{-1}\, p''\, s''$ the equivalence class of R containing s' and s'' and
- $Y = X \cap (\pi'\, q'^{-1}\, q'\, s' \cup \pi''\, q''^{-1}\, q'\, s')$

$\pi'\, s'$ is in Y and $\pi''\, s''$ in X but not in Y hence Y is a nonempty proper subset of X. There cannot be elements $x \in X$ and $y \in Y$ connected by R' because then by commutativity

$q'\, s' = q'\, \pi'^{-1}\, y = q''\, \pi''^{-1}\, x = q''\, s''$ contrary to the assupmtion. But such a partition of X contradicts that R is minimal. This handles the case with all functions defined on s' and s''. For the other cases:

- If $\operatorname{def} q' \ni s' \notin \operatorname{def} p'$ or $\operatorname{def} q'' \ni s'' \notin \operatorname{def} p''$ then setting X to the equivalence class of undef and Y as above yields the same contradiction.
- If $\operatorname{def} p' \ni s' \notin \operatorname{def} q'$ or $\operatorname{def} p'' \ni s'' \notin \operatorname{def} q''$ then setting X as above and Y to the equivalence class of undef yields a contradiction to $s' \in \operatorname{def} q'$ or $s'' \in \operatorname{def} q''$.

$$\begin{tikzcd}
Q \arrow[rd, dotted, "\exists ! q"] \arrow[rrd, "q'"] \arrow[rdd, "q''"'] & & \\
 & P \arrow[r, "p'"] \arrow[d, "p''"] & S' \arrow[d, "f'"] \\
 & S'' \arrow[r, "f''"] & S
\end{tikzcd}$$

Figure 58. The pullback.

Hence q is well defined in all cases which completes the proof of the universal property of the pushout.

By a well-known result from category theory a colimit may be constructed as a coequaliser of two coproducts. As the empty set is the zero object of <u>SETP</u> (but not in <u>SET</u>!) the coproduct equals the pushout with two zero morphisms. The proof above has shown that this is the disjoint union of the objects which exists for all possible discrete diagrams. Hence <u>SETP</u> has all coproducts furthermore a coequaliser is a pushout with S' = S". This proves cocompleteness.

For completeness first construct a pullback as in Figure 58. With the special element undef – neither element of S' nor S" - define

- P = {(s', s") ∈ (S'∪{undef}) x (S"∪undef) | (if s'∈def(f') then f'(s) else undef) = (if s"∈def(f") then f"(s") else undef)} \ {(undef, undef)}
- p': P→S' with
  - def(p') = {(s', s")∈P | s' ≠ undef}
  - p'((s', s")) = s'
- p": P→S" symmetrical

Clearly f'p' = f"p" forms a commutative square in <u>SETP</u>. And q has to get defined as

- def(q) = def(q')∪def(q")
- q(s) = ((if s∈def(q') then q'(s) else undef), (if s∈def(q") then q"(s) else undef))

Again q is uniquely determined by commutativity which proves the universal property. The existence of the limit of a general diagram is derived dually as for a colimit. ♦⇨24⇦

Lemma 5. In <u>MS</u> there are finite diagrams without a colimit or a limit.

Proof. To show an example a matrix representation is used:

- for every module or multiset we fix a base e = (e_1, e_2, ... e_n)
- every element m is uniquely written as a column vector v with m = e v. Reverse every v∈$\mathbb{Z}^n$ respectively v∈$\mathbb{N}^n$ gives an element.
- morphisms are represented by m x n matrices. n beeing the dimension of the source space and m of the destination space. The image of an element is the multiplication of the column vector from the right.
- x^t denotes the transposed of a vector or a matrix, i.e. the exchange of rows and columns.

$$\begin{tikzcd}
M \arrow[r, "f'=\begin{pmatrix}1\\2\end{pmatrix}"] \arrow[d, "f''=\begin{pmatrix}2\\1\end{pmatrix}"'] & M' \arrow[d, "p'=\begin{pmatrix}1&0\\0&1\\0&1\end{pmatrix}"] \arrow[ddr, "q'"] & \\
M' \arrow[r, "p''=\begin{pmatrix}0&1\\1&0\\0&2\end{pmatrix}"'] \arrow[drr, "q''"'] & P \arrow[dr, "\exists!q"] & \\
 & & Q
\end{tikzcd}$$

$$q=\begin{pmatrix}q' & q''\end{pmatrix}\begin{pmatrix}0&1&-1\\0&0&0\\1&0&0\\0&0&1\end{pmatrix}$$

Figure 59. A lacking pushout.

In the diagrams morphisms are indicated by their matrices. The dimensions of the object can be inferred from the matrix sizes. For example, f' maps a number x to a vector $(x, 2x)^t$. For $\mathbb{Z}$-modules (sets over integers instead over naturals as multisets) it is a pushout diagram. But, although, it is a commutative diagram in <u>MS</u> too it is not a pushout there. The morphism pair $q' = (2, 0)$, $q'' = (0, 1)$ has in the category of $\mathbb{Z}$-modules the connecting morphism $q= (2, 1, -1)$ which is missing in <u>MS</u>. We have to show that this problem cannot be fixed.

A pushout can be represented by the matrix P, the concatenation of the matrices P' of p' and P'' of p''. The columns of this matrix are $p'((1, 0)^t)$, $p'((0, 1)^t)$, $p''((1, 0)^t)$ and $p''((0, 1)^t)$. Commutativity boils down to the matrix equation

$$P\ (1, 2, -2, -1)= \mathbf{0}.$$

The morphism pair $q_1' = (1, 0)$ and $q_1'' = (0, 1)$ factorises over a q from P with

$$q_1'\ f'\ (1) = 1 = q\ p'\ f'(1) = q_1p'\ ((1, 2)^t)$$
$$q_1''\ f''\ (1) = 1 = q\ p''\ f''(1) = q_1p''\ ((1, 2)^t)$$

Hence the decomposition of $p'\ ((1, 2)^t)$ into base vectors (in <u>MS</u>) must contain exactly one base vector say e_1 with $q(e_1) \neq 0$. Further $q(e_1) = 1$ and e_1 cannot occur in the decomposition of $p'((0, 1)^t)$. By linear dependence $P\ (1, 2, -2, -1)= \mathbf{0}$ e_1 occurs also in the decomposition of $p''((0, 1)^t)$ and symmetrically it cannot occur in $p''((1, 0)^t)$. Hence any pullback in <u>MS</u> must have a line like the first one of P.

Similarly with $q_2' = (0, 1)$ and $q_2'' = (1, 0)$ we get a line like the second of P, with $q_3' = (0, 1)$ and $q_3'' = (0, 2)$ one like the third of P and with $q_4' = (2, 0)$ and $q_3'' = (1, 0)$ a fourth line equal $(2, 0, 1, 0)$. By

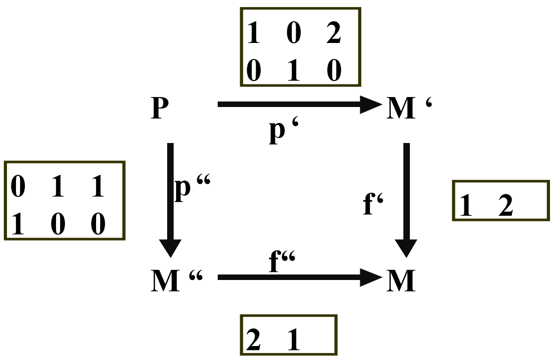
$$(2, 0, 1, 0) = (2, 1, -1)\ P$$

these four lines are linear dependent. We finish the proof by showing that such a linear dependence contradicts the universal property of the pullback.

A product aP for a row vector a corresponds to a linear dependence of the rows of P or a linear map from the module P to the integers. A linear dependence over $\mathbb{Q}$ gives an integer row vector a with $a\ P = \mathbf{0}$. The decomposition $a = a^+ - a^-$ in natural vectors gives morphisms in <u>MS</u>:

$a^+ , a^-: P \rightarrow \mathbb{N}$ with
$a^+ P' = a^+ P''$ and
$a^+ P' = a^- P''$

The uniqueness of universal connection in pushout gives $a^+ = a^-$. Hence the rows of P are linear independent.



$$\begin{tikzcd} P \arrow[r, "p'\ \boxed{\begin{matrix}1&0&2\\0&1&0\end{matrix}}"] \arrow[d, "\boxed{\begin{matrix}0&1&1\\1&0&0\end{matrix}}\ p''"'] & M' \arrow[d, "f'\ \boxed{\begin{matrix}1&2\end{matrix}}"] \\ M'' \arrow[r, "f''", "\boxed{\begin{matrix}2&1\end{matrix}}"'] & M \end{tikzcd}$$

Figure 60. A lacking limit.

The same line of argumentation shows that Figure 60 does not allow a limit ♦ ⇨24⇦

7.2.3 One-Sets

Proposition 10. 1S is left adjoint to BN. There is a natural equivalence ε: $Id_{\underline{SETP}} \rightarrow B\ 1S$ and there are isomorphisms δ_M: $M \rightarrow 1S\ B\ M$ with $B\ \delta_M = \varepsilon_{BM}$.

Proof. η : <u>1S</u>[1S S, M] $\rightarrow$ <u>SETP</u>[S, BN M]given by $\eta\ f\ s = (f_\beta\ s, f_\gamma\ s)$ is obviously a bijection between the two sets of morphisms. Naturality follows straightforward. The functor compositions of B and 1S give bijections on S respectively BM which yields the claimed properties of ε and δ. δ is a 'not natural transformation' for $Id_{\underline{1S}} \rightarrow 1S\ B$ ♦ ⇨26⇦

Proposition 11. <u>1S</u> is cocomplete and complete.

Proof. For a given diagram D in <u>1S</u> the colimit q: BN D$\rightarrow$Q exits in <u>SETP</u> by Proposition 3. Define the equivalence relation ~ on the elements of Q generated by $\sim^1$

$x \sim^1 y$ iff $\exists$ C an object of D, $\exists c \in$ BN C, $\exists \lambda \in \mathbb{N}^+$ with (BN q_C) c=x and (BN q_C) λ c = y

Let $\Theta \subseteq Q$ an equivalence class of ~ and $\Phi(\Theta)$ the set of functions compatible with multiplication:

$\Phi(\Theta) = \{f: Q \rightarrow \mathbb{N}^+ \mid \forall c \in$ BN C, C an object of D, $\lambda \in \mathbb{N}^+$ with BN $q_C c \in Q$
holds (BN q_C) λ c = λ (BN q_C) c$\}$

If $\Phi(\Theta)$ is not empty it contains a minimal function $\phi(\Theta)$ with

$\Phi(\Theta) = \mathbb{N}^+ \phi(\Theta)$

Now let P_B the equivalence classes of ~ with nonempty Φ and P the one-set with basis P_B and

p_C: C$\rightarrow$P linear with $\forall c \in$ BN C: p_C c = if $[c]_\sim \notin P_B$ then **0** else $\phi([c]_\sim)\ [c]_\sim$

Clearly these are well defined <u>1S</u> morphisms to P. Moreover they are natural transformations because the restrictions to the base skeletons are.

$$\begin{tikzcd}
D \arrow[rr, "p"] \arrow[dr, "r"'] & & P \arrow[dl, dashed, "\exists?\ r_P"] \\
& R &
\end{tikzcd}$$

To show that P has the required universal property let r: D$\rightarrow$R another natural transformation as shown in Figure 61. By the universal property of Q there is a unique r_Q: Q$\rightarrow$BN R making the diagram in <u>SETP</u> commutative. Because each r_C is a one-set morphism the elements of an equivalence class Θ of ~ are mapped either all to **0** or to the multiples of a base element b_Θ of R, the function

$r_\Theta : \Theta \rightarrow \mathbb{N}$ by r_Θ x = (r_Q x) b_Θ

is in $\Phi(\Theta)$ and r_Θ is a multiple of $\phi(\Theta)$. Hence r_Q lifts in a unique way to a <u>1S</u> morphism r_P: P$\rightarrow$R finishing the proof that P is the claimed colimit.

$$\begin{tikzcd}
BN\ D \arrow[r, "q"] \arrow[dr, "BN\ r"'] & Q \arrow[r] \arrow[d, dotted, "\exists!\ r_Q"] & BN\ P \\
& BN\ R &
\end{tikzcd}$$

Figure 61. The colimit in <u>1S</u>.

Because BN has a left adjoint it preserves limits. Hence the skeleton of the limit of a diagram D has to coincide with the limit q: Q$\rightarrow$BN D. To find a multiplication of an element r of Q with a $\lambda \in \mathbb{N}^+$ use

$\forall$ objects C of D: q_C (λ r) = if $r \in$ def q_C then λ q_C r else undefined

By the universality of Q such a λr is unique because the elements of Q are in bijection with the natural transformations from a singleton set to D. On the other hand, if x is an element of Q it corresponds to such a natural transformation and by the linearity of the morphisms in D also λr as defined above. Hence λr uniquely exists and by the same argument there is at most one s with λs = r for any $r \in q$ and $\lambda \in \mathbb{N}^+$. The base in Q is simply the set minimal elements:

B Q = $\{b \in Q \mid \forall r \in Q, \lambda \in \mathbb{N}^+$: if b = λ s then λ = 1)

We claim that any $r \in Q$ is the multiple of exactly one $b \in BQ$ and hence Q = $\mathbb{N}^+$ x BQ. Assume r = b λ = b' λ'. If $\lambda = \lambda'$ then we showed already that b = b'. If $\lambda \neq \lambda'$ then r is divisible by the least common multiple of λ and λ'. This contradicts that $b, b' \in B\ M$. Thus, r is a multiple of at most one $b \in BQ$. Such a b exists if the set $D = \{s \mid \exists \lambda \in \mathbb{N}^+ \colon \lambda\ s = r\}$ is finite. But there is a q_C that is defined on r and it maps D bijectively to $q_C\ D \subseteq \{s' \mid \exists \lambda \in \mathbb{N}^+ \colon \lambda s' = q_C\ r\}$ and the latter set is finite ♦ ⇨26⇦

7.3 Place-Transition Nets

7.3.1 Definitions and Basic Properties

Definition 12 tacitly assumes that <u>PTNET</u> is a well-defined category.

Proof: The composition of two morphisms $f\colon N \to N'$ and $g\colon N' \to N''$ is again a <u>1S</u> morphism. By

$$g\ f\ pre_N\ t = g\left(\left\{\begin{matrix} f(t) \\ pre_{N'}\ f\ t \end{matrix}\right\}\right) = \left\{\begin{matrix} g\ f\ t \\ g\ f\ t \\ pre_{N''}\ g\ f\ t \end{matrix}\right\}$$

all possible combinations end up in g f t or $pre_{N''}$ g f t as required for a <u>PTNET</u> morphism. Together with the same calculation for post this proves compositionality. The proof is only complete, because of the mentioned convention that the application of a pre or post function implies the argument to be a transition multiset. The composition is associative, because it is in <u>1S</u>. Finally the identities in <u>1S</u> are also the identities in <u>PTNET</u> ♦ ⇨27⇦

Proposition 13. A <u>PTNET</u> morphism $f\colon N \to N'$ is

- epimorphic iff f_β is surjective
- monomorphic iff $def(f_\beta) = X$ and f_β injective
- isomorphic iff f is unitary, f_β is a bijective and isolated places are mapped to places.

Proof: If f_β is surjective then clearly f is epimorphic. In the other direction we assume by contradiction in a first case $t' \in T' \setminus im(f_\beta)$. We construct a net with "doubled t'" as follows:

- $t'' \notin X'$
- $T'' = T' \cup \{t''\}$
- $pre''(t) =$ if $t \in T'$ then $pre'(t)$ else $pre'(t')$
- $post''(t) =$ if $t \in T'$ then $post'(t)$ else $post'(t')$
- $N'' = (pre'', post'')$
- $g(x') = x'$
- $h(x') =$ if $x' = t'$ then t'' else x'.

$$\begin{tikzcd} N \arrow[r, "f"'] & N' \arrow[r, shift left, "g"] \arrow[r, shift right, "h"'] & N'' \end{tikzcd}$$

Figure 62. f is epimorph iff gf=hf implies g=h

This constructs a net N'' and different morphisms g and h with g f = h f in contradiction to the definition of epimorphic.

The second case is a place p' instead of t'. If p' is not isolated it is connected to a transition t' that neither can be in $im(f_\beta)$ and the first case applies. Otherwise take N'' with two places and no transitions. g and h are defined only on p' and map it to the first respectively the second place of N''. As above this is a contradiction to f epimorphic. Hence there is a contradiction in all cases.

For the second claim it suffices to proof that if f is monomorphic then f_β fulfils the two claimed conditions. If def(f_β) is not the whole X we find as above a $y \in X$ such that

$$\begin{tikzcd}
N'' \arrow[r, shift left, "g"] \arrow[r, shift right, "h"'] & N \arrow[r, "f"'] & N'
\end{tikzcd}$$

Figure 63. f is monomorph iff fg=fh implies g=h

  g(y) = if y=x then undef else y

is a morphism N→N. Together with id_N it contradicts f to be monomorphic. For the second condition left $f_\beta(x_g) = f_\beta(x_h)$ for $x_g \neq x_h \in X$. From the net N" with a single place p" and no transition there are two different morphisms

  $g(p'') = f_\gamma(x_h)\, x_g$

  $h(p'') = f_\gamma(x_g)\, x_h$

with gf = hf in contradiction to monomorphic.

Finally, if f is iso it must be epi and mono. And f_γ must equal 1, because this is the only number with a reciprocal within the naturals. But this is not sufficient: Figure 5 shows a morphism that fulfils all these conditions but its inverse is not a morphism. If a place is mapped to a transition there exists no inverse because

- if the transition is not isolated then by bijectivity and commutativity with pre and post
- if the transition is isolated then the inverse must map it to an isolated transition too.

For the reverse direction non-isolated transition must map to transitions – otherwise the pre and post sets would also map in the image of the transition, contradicting bijectivity. Consequently non-isolated places map to places. By the above and the last condition this is also true for isolated elements. All together this ensures that the inverse of f also commutes with pre and post and hence is a PTNET morphism. This proves the last claim.

Remark. The proposition still holds for morphisms disallowed to map places to transitions. The proof has to be adapted for the part about monomorphisms. If x_g and x_h are both transitions and the pre and post sets are 1:1 a net with a similar transition (with pre and post weights all 1) can map to both transitions and hence replace the single place net above. ♦ ⇨28⇦

Proposition 15. PTNET: 1S→PTNET is the left adjoint of U. Moreover, they form a coreflection.

Proof: The bijection between

  PTNET [PTNET M, N'] → 1S [M, U N']

translates (by setting M = 1S P and U N' = 1S X') to

  PTNET [PTNET 1S P, N'] → 1S [1S P, 1S X']

Obviously the interpretation the same morphism once in PTNET and once in 1S is natural. The unit given by

  Id ∈ PTNET [PTNET M, PTNET M] → ε_M ∈ 1S [M, U PTNET M]

is clearly an isomorphism and hence the adjunction a coreflection ♦ ⇨29⇦

7.3.2 Clustering

Proposition 17. Let f: N→N', $K_f = X_N \backslash def(f_\beta)$ and $S' \subseteq X'$. Then

- $def(f_\beta)$ is transition-bordered
- K_f is place-bordered,
- if S' is place-bordered then also $f_\beta^{-1}(S') \cup K_f$
- if S' is transition-bordered then also $f_\beta^{-1}(S')$
- if S' is non-splitting then also $f_\beta^{-1}(S')$ relative $def(f_\beta)$

Proof: Let $p \in {}^\bullet t$ and $t \notin def(f_\beta)$ then f t = **0** and by the definition of a PTNET morphism f pre t = **0**. Thus f p = **0** and equivalently $p \notin def(f_\beta)$. With the similar argument for post this signifies that no transition outside of $def(f_\beta)$ is connected to the inside hence $def(f_\beta)$ does not have border places and is transition-bordered.

Let S' be transition-bordered, $S = f_\beta^{-1}(S')$ and $p \in {}^\bullet t \cap S$. Clearly f p ≠ **0**. If f pre t = pre f t then f t∈S' because f p is and S' is transition-bordered. Otherwise f pre t = f t and f t∈S' because f p is. In both cases f t∈S' yields t∈S and S is transition-bordered.

If S' is place-bordered, then X'\S' is transition-bordered and by the previous also $f^{-1}(X'\backslash S') = f^{-1}(X'\backslash S') = def(f_\beta) \backslash f^{-1}(S')$. Hence the complement is place-bordered.

The last claim is a direct application of the definitions ♦ ⇨30⇦

7.3.3 Net Invariants

Proposition 19. Let f: N→N' a morphism of place-transition nets. Then

- if i': $X_{N'} \to \mathbb{Z}$ is a place invariant of N' then $i = \sum_{p \in P} i' f p$ is a place invariant of N.
- Semi-positive place invariants of N are 1 to 1 with morphisms to single place nets.
- if $j \in 1S\ T_N$ is a transition invariant of N then $j' = \sum_{\substack{t \in T \\ f\,pre\,t \neq f\,t}} j(t)\,f(t) = \sum_{\substack{t \in T \\ f\,post\,t \neq f\,t}} j(t)\,f(t)$ is a transition invariant of N'.
- A semi-positive transition invariant corresponds to a unitary folding from a T-system (a PTNET with all arc cardinalities 1 and $|{}^\bullet p| = 1 = |p^\bullet|$ for all places).

Proof: Let i be a place invariant and i_X be the linear function $1S\ X \to \mathbb{Z}$ given by
$i_X(x)$ = if x∈P then i(x) else i pre(x).
Because i is a place invariant for each transition t holds
$i_X\, t = i_X\, pre\ t = i_X\, post\ t$.
We call a linear function fulfilling this equation an extended place invariant. Clearly mapping i to i_X is a bijection from place invariants to extended place invariants. Equivalent to the first claim is i_X f is an extended place invariant. But this is proved in the same way as the compositionality of morphisms (for Definition 12, the absence of negative coefficients has not been used). For the second claim we only need to identify $\mathbb{N}$ with the net consisting of a single place PTNET $\mathbb{N}$ which turns the semi-positive invariant i_X into a net morphism.

Hence a P-invariant can be lifted from N' to N. But surprisingly the propagation in the direction of the morphism may fail as Figure 64 shows.

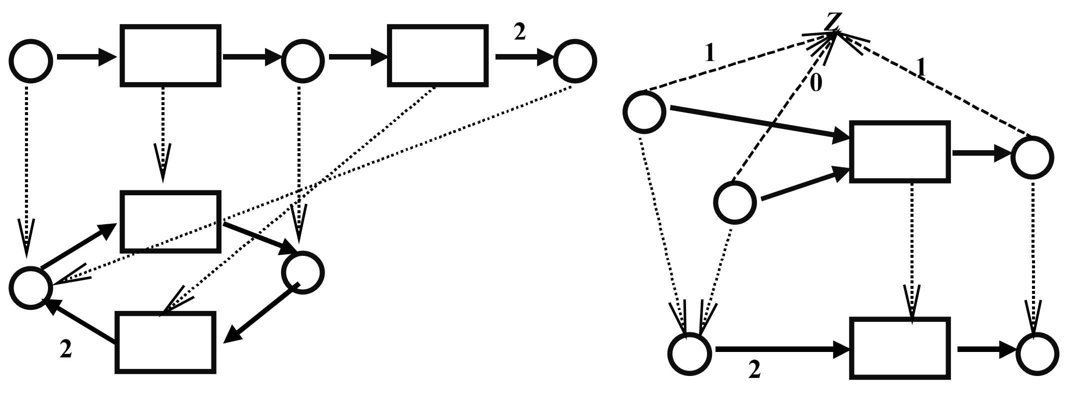


Figure 64 A place invariant may not transfer because of global (left) or local (right) incompatibility.

For the third claim the following is derived:

$$\mathbf{0} = (\text{post-pre})\; j$$

$$\mathbf{0} = f\,(\text{post-pre})\; j$$

$$\mathbf{0} = f\,(\text{post-pre})\;\Big(\sum_{\substack{f\,pre\,t = f\,t\\ f\,post\,t = f\,t}} j(t) + \sum_{\substack{f\,pre\,t \neq f\,t\\ f\,post\,t = f\,t}} j(t) + \sum_{\substack{f\,pre\,t = f\,t\\ f\,post\,t \neq f\,t}} j(t) + \sum_{\substack{f\,pre\,t \neq f\,t\\ f\,post\,t \neq f\,t}} j(t) \Big)$$

$$\mathbf{0} = f\,(\text{post-pre})\;\Big(\sum_{\substack{f\,pre\,t \neq f\,t\\ f\,post\,t = f\,t}} j(t) + \sum_{\substack{f\,pre\,t = f\,t\\ f\,post\,t \neq f\,t}} j(t) + \sum_{\substack{f\,pre\,t \neq f\,t\\ f\,post\,t \neq f\,t}} j(t) \Big)$$

$$\mathbf{0} = f \sum_{\substack{f\,pre\,t \neq f\,t\\ f\,post\,t = f\,t}} j(t) \;-\; f \sum_{\substack{f\,pre\,t = f\,t\\ f\,post\,t \neq f\,t}} j(t) \;-\; pre\; f \sum_{\substack{f\,pre\,t \neq f\,t\\ f\,post\,t = f\,t}} j(t) \;+\; post\; f \sum_{\substack{f\,pre\,t = f\,t\\ f\,post\,t \neq f\,t}} j(t) \;+\; (\text{post-pre})f \sum_{\substack{f\,pre\,t \neq f\,t\\ f\,post\,t \neq f\,t}} j(t)$$

The first two summands are transition multisets the last tree place multisets. Hence already the first two terms vanish and their equality implies

$$j' = f \sum_{f\,pre\,t \neq f\,t} j(t) = f \sum_{f\,post\,t \neq f\,t} j(t) \;\text{ and}$$

$$\mathbf{0} = (\text{post} - \text{pre})\; j'$$

which is the third claim.

For the fourth claim T clearly is a transition invariant of a T-system and by the previous a folding maps the transitions of a T-System to a transition invariant. For the reverse direction let j a transition invariant of a net N and let

$$T_J = \{(t, \alpha) \in T_N \times \mathbb{N}^+ \mid 1 \leq \alpha \leq j(t)\}$$

$$P_J = \{(t, p, \beta) \in T_J \times P_N \times \mathbb{N}^+ \mid 1 \leq \beta \leq pre\; t\; p\}$$

$$Q_J = \{(t, p, \beta) \in T_J \times P_N \times \mathbb{N}^+ \mid 1 \leq \beta \leq post\; t\; p\}$$

b_p: $\{(u, p, \gamma) \in Q_J\} \rightarrow \{(t, p, \beta) \in P_J\}$ bijective for $p \in P_N$

$$pre_J\,(t, p) = \sum_{(t,p,\beta) \in P_J} (t, p, \beta)$$

$$post_J\,(t, p) = \sum_{(u,p,\gamma) \in Q_J} b_p(u, p, \gamma)$$

$$J = (pre_J, post_J)$$

$$f\colon J \rightarrow N \text{ with } f(t, \alpha) = t \text{ and } f(t, p, \beta) = p$$

The bijections b_p exist for every place p of N because j is a transition invariant which is equivalent with pre j = post j. By construction of f pre_J = pre_N f and f $post_J$ = $post_N$ f hence f is

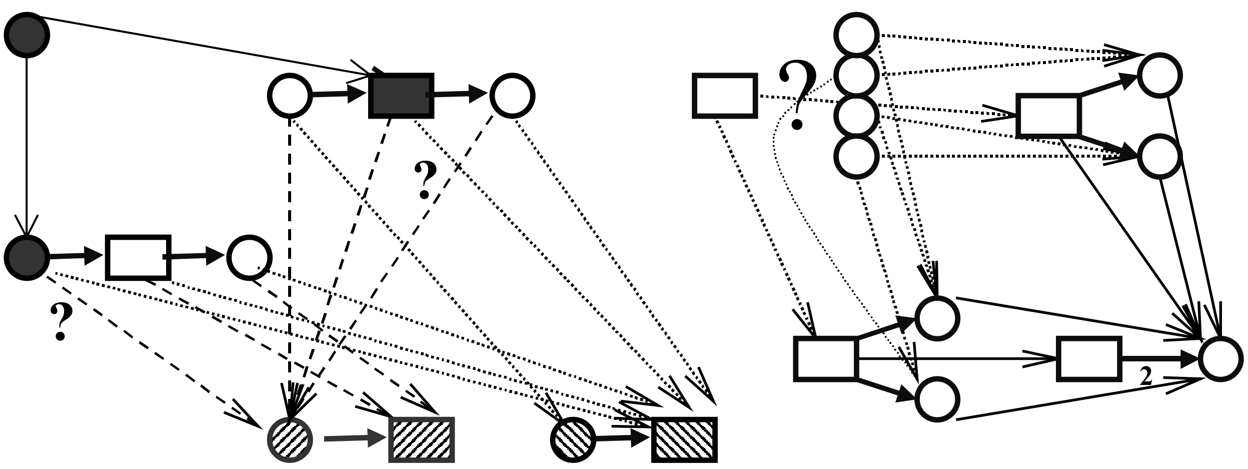


Figure 65. Diagrams without pushout (left) and without pullback(right).

a morphism and even a folding. J is a transition system because $|{}^\bullet p| = 1 = |p^\bullet|$ is a consequence of the b_p beeing bijective ♦ ⇨31⇦

Remark 20. The major limitations of PTNET are the following: universal constructions do not exist in general and behaviour transfer is complicated.

The category PTNET has several limitations. Although pushouts exist for foldings neither pushouts nor pullbacks exist in general. By commutativity the black elements on the left side of Figure 65 have to map to the same node of the pushout. Should it be a transition or a place? For a factorisation to the left bottom net to a place but for the right bottom net to a transition!
Because morphisms may mix transitions and places arbitrarily the bipartite structure does not transfer. Furthermore for pullbacks additionally the pre and post functions may clash.

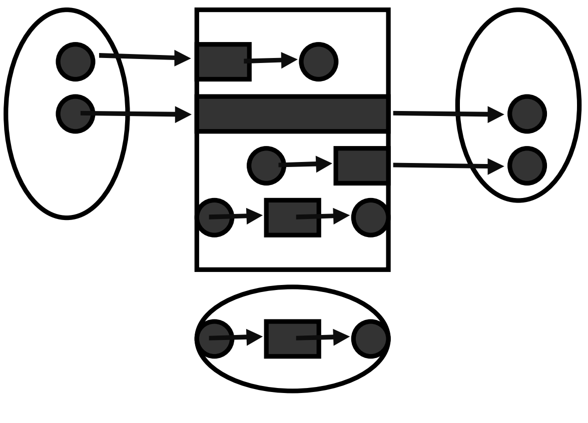


Figure 66. The possible images of a transition.

To see this remember that the underlying functor U: PTNET→1S has a left adjoint (Proposition 15) so it preserves limits by a well-known result from category theory. Hence the right hand side of Figure 65 shows the nodes of a pullback if it would exists. But there is no way to define pre and post allowing both connecting morphisms.

Finally the transfer of behaviour gets complicated. As shown in Figure 66 there are five possible combinations to map a transition and its neighbouring places - and these five cases have to be reconsidered for every proof. ♦ ⇨31⇦

7.3.4 From Clustering to Folding

Lemma 22. Let $\Delta x = (x,x)$ and pp = (pre, post) ∈ MS(MS T, MS P) x MS(MS T, MS P) ≅ MS(MS T, MS P x MS P). For a PTNET morphism f: N→N' the following properties are equivalent:

- f is place-preserving

- ($\forall t\in T$: f pp t = pp' f t or f pp t = Δ f t) and
  ($\forall p\in P\cap$def f_β: $\exists p'\in P$ with f_β $p'\in P'$ such that p and p' are connected by an undirected path with all nodes in def(f_β)).

Proof: The cartesian product of two multisets equals the multiset over the disjoint union of the bases and this equals the product as universal construction. Hence a pair of morphisms MS T$\rightarrow$MS P gives a universal connecting morphism MS T$\rightarrow$MS P x MS P which is the claimed isomorphism in the definition of pp.

Assume f place-preserving and $t\in T$. If f pre t = pre f t then f t is a transition and post f t = f t would imply that the places of $t^\bullet$ map to the transition $f_\beta t$ which contradicts place-preserving. Hence post f t = post f t and f pp t = pp' f t. Otherwise if f pre t = f t then place-preserving implies that f pre t is either **0** or a multiple of a place. By the PTNET morphism definition both cases result in f post t = f t and f pp t = Δf t. Hence the first property includes the second.

Reverse, assume the second and let

Q = $\{p\in P \mid f_\beta p\in P'\}$

Any place of N not in Q is connected to a place in q and on this path there is a transition $t\in T$ with $^\bullet t^\bullet \neq {^\bullet t^\bullet}\cap Q \neq \{\}$ because each connection component of X contains a place in Q. If f pp t = pp f t then f_β t is a transition and $^\bullet t^\bullet\subseteq Q$. If f pp t = Δ f t then all places of $^\bullet t^\bullet$ are mapped to the same node as t and $^\bullet t^\bullet$ is either a subset of Q or disjoint from it. Both cases result in a contradiction hence f is place-preserving ♦ ⇨32⇦

Proposition 23. The forgetful functor U has a right adjoint PP: PTNET$\rightarrow$PPNET.

Proof: The basic idea is, to replace every transition by three transitions and a place as shown in Figure 7. This deals with the five combinations of the image of a transition and its pre and post sets for a PTNET morphism (see Figure 66) and is defined on a net N as follows:

- four new symbols
  - d for direct,
  - i for input,
  - o for output and
  - n for internal or inherited
- $P_{PP} = \{x^n \mid x \in X\}$
- $T_{PP} = \{t^y \mid t\in T, y\in \{d,i,o\}\}$
- the x^y notation is expanded to multisets in the natural way
- $\text{pre}_{PP}\, t^y$ = if y = o then t^n else $(\text{pre } t)^n$
- $\text{post}_{PP}\, t^y$ = if y = i then t^n else $(\text{post } t)^n$
- PP N = (pre_{PP}, post_{PP}: MS $T_{PP}\rightarrow$MS P_{PP})

$$\begin{tikzcd}
\mathbf{N_{PP}} \arrow[r, "\varepsilon_N"] & \mathbf{N} \\
\mathbf{N'} \arrow[u, dashed, "\exists!\, g"] \arrow[ur, "f"'] &
\end{tikzcd}$$

Figure 67. The unique factorisation g.

Clearly PP N is a net. Furthermore,

- $\varepsilon_N\, x^y = x$: $N_{PP}\rightarrow N$

define a PTNET morphism ε_N: $N_{PP}\rightarrow N$. An f: N'$\rightarrow$N factorises with a unique place-preserving morphism g throug ε_N. g is defined on $x'\in X'$ by

$$
g(x') = \begin{cases}
\text{if } x' \in P' \text{ or } f\,pre'\,x' = f\,x' = f\,post'\,x' & \text{then } (f\,x')^n \\
\text{else if } f\,pre'\,x' = f\,x' \text{ and } f\,post'\,x' = post\,f\,x' & \text{then } (f\,x')^o \\
\text{else if } f\,pre'\,x' = pre\,f\,x' \text{ and } f\,post'\,x' = f\,x' & \text{then } (f\,x')^i \\
\text{else if } f\,pre'\,x' = pre\,f\,x' \text{ and } f\,post'\,x' = post\,f\,x' & \text{then } (f\,x')^d
\end{cases}
$$

Clearly g is place-preserving. The proof that it is a morphism, needs to check each case of the definition. For an input transition t':

$g\,t' = (f\,t')^i$

$g\;pre\;t' = (f\;pre'\;t')^n = (pre\;f\;t')^n = pre\;g\;t'$

$g\;post\;t' = (f\;post\;t')^n = (f\,t')^n = post\;g\;t$

and similarly in the three other cases.

To prove the unicity of g notice that the commutativity of the triangle requires $g\,x' = (f\,x')^y$. If x' is a place y = n because g is place-preserving. If x' is a transition there is only one possibility to select y such that g fulfils the place-preserving-morphism-property on $\{x'\} \cup {}^\bullet x' \cup x'^\bullet$, namely the one given above in the definition of g.

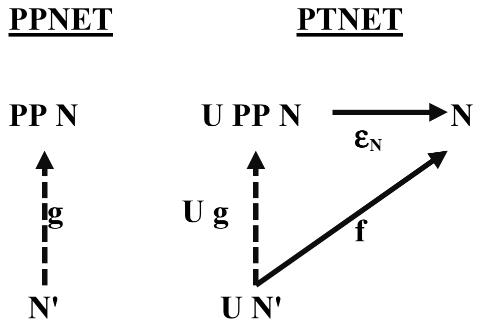

$$\begin{tikzcd}
\underline{\textbf{PPNET}} & \underline{\textbf{PTNET}} & \\
\textbf{PP N} & \textbf{U PP N} \arrow[r, "\varepsilon_N"'] & \textbf{N} \\
\textbf{N'} \arrow[u, dashed, "g"'] & \textbf{U N'} \arrow[u, dashed, "\textbf{U g}"] \arrow[ur, "\textbf{f}"'] &
\end{tikzcd}$$

Figure 68 The unique factorisation g as universal arrow from U to N.

If one redraws Figure 67 as Figure 68 one sees that ε_N is a universal arrow from U to N which by a result from category theory ([Lan71], theorem 2, page 81) yields that PP is a functor and that it is right adjoint to U with the counit ε ♦ ⇨32⇦

Proposition 24. Let PL: <u>PPNET</u>→<u>PPNET</u> be the functor dropping all transitions and keeping the places of a net. U PL: <u>PPNET</u>→<u>1S</u> forms a coreflection with the left adjoint PP PTNET and forms a reflection with the right adjoint MM2: <u>1S</u>→<u>PPNET.</u>

Proof. Let M be an object of <u>1S</u> and N of <u>PPNET</u>. PTNET M has only places, so PP has no transitions to expand and produces an isomorphic object. Furthermore,

<u>PPNET</u>[PP PTNET M, N] ≅ <u>1S</u>[M, U PL N]

is just a reinterpretation of the same function once as a place-preserving morphism and once as <u>1S</u> morphism between M and the multiset over the places of N. Clearly the units of this adjunction are isomorphisms in <u>1S</u>.

The construction of the claimed functor MM2 uses the same categorical technique as Proposition 23 (see Figure 68). To a given one-set M = 1S S a net N = MM2 M contains a node for each minimal combination of pre and post

$$X = \{(m', m'') \in M \times M \mid \forall l', l'' \in M, \lambda \in \mathbb{N}^+: \text{ if } (m', m'') = (\lambda l', \lambda l'') \text{ then } \lambda = 1 \text{ or } m' = m'' = 0\}$$

The set of places is an embedding of S in X

$$P = \{(1s, 1s) \in X \mid s \in S\}$$

and pp is defined on a transition $(m', m'') \in T = X \setminus P$ by

$$pp\,(m', m'') = \Big(\sum_{s \in S} m'(s)\,(1s, 1s),\; \sum_{s \in S} m''(s)\,(1s, 1s) \Big).$$

The counit ε_M: U PL MM2 M→M is simply the projection with

$\varepsilon_M\,(1s, 1s) = s$ for $(1s, 1s) \in P$

Now, let f: U PL N' $\rightarrow$ M be a <u>1S</u> morphism. If there exists a factorisation f = ε_M U PL g with a place-preserving morphism g it fulfils

$g\ p' = (f_\gamma\ p')\ (1\ f_\beta p',\ 1\ f_\beta p')$ for $p' \in P'$

$g\ t' = \lambda_t'\ ((1/\lambda_t')\ g\ pp'\ t')$ for $t' \in T'$ and an appropriate $\lambda_t' \in \mathbb{N}^+$

By the definition of X such a λ_t' is uniquely determined unless f pp' t' = **0**. These equations completely determine g, hence, g is unique. Conversely, these equations define a linear function g which maps places to places and yields the wanted factorisation. That g is indeed a <u>PPNET</u> morphism is derived by:

$$g\ pp'\ t' = \left(\sum_{s \in S} (f\ pre'\ t')\ (s)\ (1s, 1s),\ \sum_{s \in S} (f\ post'\ t')\ (s)\ (1s, 1s) \right) = \begin{cases} \Delta\ g\ t' \\ pp\ (f\ pp'\ t') = pp\ (g\ t') \end{cases}$$

Thus ε_M: U PL MM2 M $\rightarrow$ M is an universal arrow from U PL to M which yields the claimed adjunction ♦ ⇨33⇦

Proposition 25. <u>PPNET</u> is cocomplete and complete.

$$\begin{tikzcd}
N \arrow[r, "f'"] \arrow[d, "f''"'] & N' \arrow[d, "u'"] \arrow[ddr, "q'"] & \\
N'' \arrow[r, "u''"] \arrow[drr, "q''"'] & U \arrow[dr, dotted, "\exists!q"] & \\
 & & Q
\end{tikzcd}$$

Figure 69. Definition of a pushout.

Proof: Figure 70 shows how to construct a pushout in <u>PPNET</u> from a pushout u', u'', U in <u>1S</u>.

$P_U = u_\beta'\ P' \cup u_\beta''\ P''$ and

$T_U = B\ U \backslash P_U$

defines a bipartite structure on U such that u' and u'' are place-preserving. Define projections

$\pi'\ x'$ = if $x' \in T'$ and $u'_\beta\ x' \in T_U$ then x' else **0**

$\pi''\ x''$ = if $x'' \in T''$ and $u''_\beta\ x'' \in T_U$ then x'' else **0**

For a $t \in T$ with $f_\beta'\ t \in T'$ and $f_\beta''\ t \in T''$ we get

u' pp' π' f' t = u' pp' f' t = u' f' pp t = u'' f'' pp t = u'' pp'' f'' t = u'' pp'' π'' f'' t

Because all other nodes of N are mapped to zero it follows

u' pp' π' f' = u'' pp'' π'' f'' t

Hence there is a universal connection pp_U from U to MS P_U.

To be precise u' pp' π' is not a 1S morphism but a pair of multiset morphism. To fix this one uses the projections (p^*) from $MS^2\ P_U$ to $\mathbb{N}$ which evaluate either the pre or the post multiset at a fixed place p. The composition with the (p^*) are 1S morphisms and there is a unique pp_{p^*} and pp_U is the formal sum of all pp_{p^*}. pp_U is unique because the composition with all p^* is unique.

$$\begin{tikzcd}
U\ N \arrow[r, "U\ f'"] \arrow[d, "U\ f''"'] & U\ N' \arrow[d, "u'"] \arrow[ddr, "u'pp'\pi'"] & \\
U\ N'' \arrow[r, "u''"] \arrow[drr, "u''pp''\pi''"'] & U \arrow[dr, dotted, "\exists!pp_U"] \arrow[dd, dotted, "\exists!pp_{p^*}"] & \\
 & & MS^2\ P_U \arrow[dl, "(p^*)"] \\
 & N &
\end{tikzcd}$$

Figure 70. A pushout in <u>1S</u> and <u>PPNET</u>.

The restriction of pp_U to T_U gives a net $N_U = pp_U : T_U \rightarrow MS^2\ P_U$ and clearly u' and u'' are connecting place-preserving morphisms.

For commutative q' and q" there is a unique q∈<u>1S</u>[U, Q] by the universal property of U. It suffices to verify that q is a place-preserving morphism. For a $p \in P_U$ there is a $p' \in P'$ with $u'_\beta\, p' = p$ or a similar p". As q' is place-preserving it maps p' to a place (or **0**) and by commutativity q maps p to the same place. Hence q is place-preserving.
If q is defined on $t \in T_U$ there is a $t' \in T_N'$ with q' defined on t' and $q_\beta' = q_\beta\, u_\beta'$ or the analogous for a t". Thus

$$q\ pp\ t = (1/u_\gamma' t')\ q\ u'\ pp\ t = (1/u_\gamma' t')\ q'\ pp\ t = (1/u_\gamma' t') \begin{Bmatrix} pp\ q'\ t \\ q'\ t \end{Bmatrix}$$

$$= (1/u_\gamma' t') \begin{Bmatrix} pp\ q\ u'\ t \\ q\ u'\ t \end{Bmatrix} = \begin{Bmatrix} pp\ q\ t \\ q\ u\ t \end{Bmatrix}$$

This finishes the proof that (pp_U) is the pushout in <u>PPNET</u>. Again the coproducts are simply the disjoint unions and exist for every diagram without morphisms. Cocompleteness follows by the well-known fact from category theory that every colimit reduces to a coequaliser (which is a special pushout) of coproducts.

Figure 71 shows the proof of completeness. Let v', v", V the pullback in <u>1S</u> and

$P_U = \{p \in B\ V \mid v_\beta' p \in P' \text{ and } v_\beta'' p \in P''\}$
$T_V = B\ V \setminus P_U$
$T_U = \{(t_v, y) \mid t_v \in T_V,\ y \in (1S\ P_U,\ 1S\ P_U)$ with
  $v'\ y = $ if $v'_\beta\, t_v \in P'$ then $\Delta\ v'\ t_v$ else $pp'\ v'\ t_v$ and
  $v''\ y = $ if $v''_\beta\, t_v \in P''$ then $\Delta\ v''\ t_v$ else $pp''\ v''\ t_v$ }
$pp_U\,(t_v, x) = x$

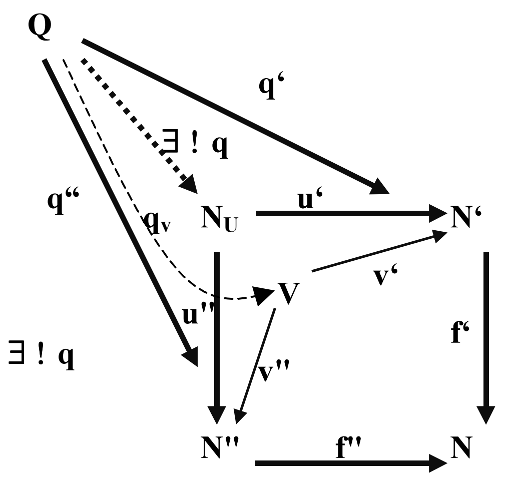


Figure 71. A pullback in <u>PPNET</u>.

Clearly $N_U = pp_U$ is a net and
  $u'\ x = $ if $x \in P_U$ then $v'\ x$ else if $x = (t_v, y) \in T_U$ then $v'\ t_v$
  $u''\ x = $ if $x \in P_U$ then $v''\ x$ else if $x = (t_v, y) \in T_U$ then $v''\ t_v$
defines place-preserving morphisms $N_U \rightarrow N'$ respectively $N_U \rightarrow N''$ making the square commutative. To proof the universal property let q', q", Q a connecting diagram and q_v the universal connection to V in <u>1S</u>. If a connecting q exists it must fulfil
  $q\ x = $ if $q_{v\ \beta}\ x \in P_U$ then $q_v\ x$ else $(q_v\ x,\ q_v\ pp\ x)$
by commutativity and the definition of a morphism. If a connecting q exists it is uniquely determined by the above equation. But does q exist? If $q_{v\ \beta}\ x \notin P_U$ then either $q'_\beta x$ or $q''_\beta x$ is a transition and hence also x. Because q_V is natural
  $v'\ (q_v\ pp\ x) = q'\ pp\ x = $ if $q'_\beta\, x \in P'$ then $\Delta\ q'\ x$ else $pp'\ q'\ x$
    $= $ if $v'_\beta\ (q_v\, x) \in P'$ then $\Delta\ v'\ (q_v\ x)$ else $pp'\ v'\ (q_v\ x)$
  $v''\ (q_v\ pp\ x) = q''\ pp\ x = $ if $q''_\beta\, x \in P''$ then $\Delta\ q''\ x$ else $pp''\ q''\ x$
    $= $ if $v''_\beta\ (q_v\, x) \in P''$ then $\Delta\ v''\ (q_v\ x)$ else $pp''\ v''\ (q_v\ x)$

Thus $(q_v\ x)$ and $(q_v\ pp\ x)$ are multiples of a transition from T_U, q is well defined and the uniqueness argument from above shows that it is a place-preserving morphism. Hence N_U is the pullback.

A product of two objects equals the pullback with zero morphisms and clearly the above construction generalises to products of arbitrary sets of nets. Hence PPNET is complete ♦ ⇨33⇦

7.3.5 Folding Nets

Proposition 27. The underlying functor U has a right adjoint F: PPNET→FNET.

Proof. F N is constructed according to Figure 9:

- two new symbols q and r
- $P_F = \{p^q \mid p\in P_N\}$
- $T_F = \{x^r \mid x\in X_N\}$
- $pp_F\ x^r =$ if $x\in T_N$ then $(pp_N\ x)^r$ else (x^r, x^r)
- $\varepsilon_N\ x^y = x$ for $y\in \{r, q\}$

Again $F\ N = N_F$ is a net, ε_N a place-preserving morphism and there is a unique factorisation

$g\ (x) =$ if $f_\beta\ x =$ undef then undef else if $f_\beta\ x\in\ P'\ x^q$ else x^r

This is the unique way to define a commutative folding and the rest of the proof exactly works as for PP (Proposition 23) ♦ ⇨34⇦

Proposition 28. FNET is cocomplete and complete.

Proof: The pushout is the same as in PPNET the universal connection q automatically becomes a folding if q' and q'' are foldings. The pullback is different: transitions have to get paired only with transitions. With this change the same construction and proof works ♦ ⇨34⇦

7.4 High-Level Nets

7.4.1 Coloured Nets

Proposition 31. src and dst extend to functors SRC and DST: C*NET→*NET. DST is left adjoint to ID (with ID N = Id_N), forming a reflection, and SRC is right adjoint to ID, forming a coreflection. The functors PP and F and their adjunctions with U lift to the categories C*NET and commute with the former adjunctions.

Proof: Clearly, SRC $f = f^{\,s}$ and DST $f = f^{\,d}$ both define functors. $\eta_{N,C}$: [ID N, C] ≅ [N, SRC C] defined by $\eta\ k = k^s$ is bijective. Naturality and adjointness are derived by direct application of the definitions and may be read from Figure 72. The unit ε_N : N→SRC ID N is clearly an isomorphism, so that the adjunction is a coreflection. The proof for DEST is analogous.

Purely categorical arguments show that SRC Q' = Q and DST Q' = Q lifts a functor Q to a functor Q' in the comma category. This construction also lifts adjunctions. Because PP and F map foldings to foldings, these functors are lifted to the coloured categories and this construction clearly commutes with SRC, DST and ID ♦ ⇨36⇦

$$\begin{tikzcd}
N & C^s \arrow[r, "C"] \arrow[d, "g^s"'] & C^d \arrow[d, "g^d"] \\
N' \arrow[u, "f"] & C'^s \arrow[r, "C'"] & C'^d
\end{tikzcd}
\qquad
\begin{tikzcd}
k=(h,\ C\ h\ Id) \arrow[rr] \arrow[ddd] & & h \arrow[ddd] \\
{[Id\ N,\ C]} \arrow[r, "\eta_{N,X}"] \arrow[d] & {[N,\ SRC\ C]} \arrow[d] & \\
{[Id\ N',\ C']} \arrow[r, "\eta_{N',X'}"'] & {[N',\ SRC\ C']} & \\
(g^s\ h\ f,\ g^d\ C\ h\ f) \arrow[rr] & & g^s\ h\ f
\end{tikzcd}$$

*Figure 72. Adjointness proof in C*NET.*

Proposition 32. CPPNET and CFNET both are cocomplete and complete.

Proof: Let D be a diagram in CPPNET or CFNET, u^s: SRC D$\rightarrow U^s$ and u^d: DST D$\rightarrow U^d$ be the colimits in *NET which exist because these two categories are cocomplete as shown in Figure 73. Because the $u_C{}^d$ C: DST C$\rightarrow U^d$ for the coloured nets C of D form a natural transformation SRC D$\rightarrow U^d$ there is a unique commutative U: $U^s\rightarrow U^d$. As DST has a right adjoint, it preserves colimits so we had better not spoil anything there. Instead, we construct a morphism u^f: $U^s\rightarrow U^f$ such that U factorises into a unitary folding d: $U^f\rightarrow U^d$. Let

- ~ be the minimal equivalence relation generated by $\{(t, p) \in T^s x P^s \mid U_\beta\ p = U_\beta\ t \neq \mathbf{0}$ for $p \in {}^\bullet t^\bullet\}$.
- T^f be the equivalence classes of ~ whose elements are mapped by U_β to a transition.
- P^f be the equivalence classes of ~ whose elements are mapped by U_β to a place.
- $u^f = (u^f{}_\beta, U_\gamma)$: $U^s\rightarrow U^f$ with
- $u^f{}_\beta\ (x)$ = if U x = **0** then undefined else ~ equivalence class of x
- $pp^f = pp^s\ (u^f)^{-1}$
- d: $U^f\rightarrow U^d$ with d u^f x = U x

Because U is a place-preserving morphism, $u^f{}_\beta$ is injective on $(u^f{}_\beta)^{-1}$ T^f which implies that pp^f is well defined and d is a unitary folding. u^f $u_C{}^s$ and $u_C{}^d$ form a natural transformation D$\rightarrow$d in CPPNET and CFNET respectively.

The claim is that d is already the wanted colimit. Given a natural transformation q: D$\rightarrow$Q by the universality of U^s and U^d, there exist commutative connections $q_U{}^s$: $U^s\rightarrow Q^s$ and $q_U{}^d$: $U^d\rightarrow Q^d$. Commutativity and the universality of U^s yield Q $q_U{}^s$ = $q_U{}^d$ U and hence q_U. Because of this and because Q is a unitary folding, $q_U{}^s$ factorises through q^f u^f for an appropriate q^f. Hence these q^f and q^d form a connecting colour morphism d$\rightarrow$Q. Also, this connecting morphism is unique because U^s and U^f are universal and u^f is epimorphic. Hence d is the claimed colimit.

$$\begin{tikzcd}
C^s \arrow[r, "u_C{}^s"] \arrow[rrr, bend left, "q_C{}^s"] \arrow[d, "C"'] & U^s \arrow[r, "u^f"] \arrow[rr, dotted, bend left=50, "\exists!q^s"] \arrow[d, dotted, "\exists!U"'] & U^f \arrow[r, dotted, "q^f"] \arrow[dl, "d"'] & Q^s \arrow[d, "Q"] \\
C^d \arrow[r, "u_C{}^d"'] \arrow[rrr, bend right, "q_C{}^d"'] & U^d \arrow[rr, dotted, "\exists!q^d"] & & Q^d
\end{tikzcd}$$

Figure 73. The construction of a colimit

The situation for limits is different. N → (0: 0→N) which maps a net to the zero morphism to this net is a functor yielding an adjunction. Thus if the limit of a diagram D exists it must equal the commutative connection U from the limit of SRC D to the limit of DST D. It remains to be shown that U is a unitary folding.

If U were to map a transition $t \in T^s$ to a place, all the u_C^s would also do so. However, this would contradict the universal property of U^s: a single place net mapping its unique place to u_C^s t would not factorise through U^s. Hence U is a folding. Similarly, a contradiction is derived first for a place p with $U_\gamma\, p \neq 1$ and finally for a transition t with $U_\gamma\, t \neq 1$ ♦ ⇨36⇦

7.4.2 Hierarchical Nets

7.4.3 Iterated Couniversal Constructions

Proposition 35. A cocomplete category possesses all maximal ι reductions.

Proof: Let D be the diagram consisting of all epimorphisms c: N→C such that c factorises every ι reduction of N. The colimit R of D exists and has the required couniversal property. It remains to be shown that r: N→R is a ι reduction.

Assuming the contrary, there are f, f'∈I with rf ≠ rf' and (r f, r f')∈ι (refer to Figure 74). Let v: R→V the colimit (called coequaliser) of the diagram consisting of rf and rf'. A reduction q: N→Q factorises to q = q' r and hence (q f, q f') = (q' r f, q' r f')∈ι, and because Q is an ι reduction, q' r f equals q' r f'. By the universal property of the coequaliser V, there exists a commutative connection V→Q. Thus vr: N→V factorises every ι reduction, it is contained in the diagram D and there exists a v': V→R with r = v' v r. But vrf = vrf' implies v'vrf = v'vr f' and r f = r f', which contradicts the assumption rf ≠ rf' ♦ ⇨39⇦

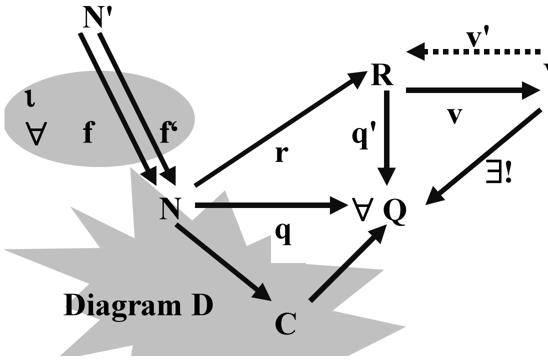


Figure 74. The existence proof.

Proposition 36. If ι is a relation on morphisms as in Definition 34 and κ is the least relation fulfilling points (i) to (iii) from below then an r is an ι reduction of N iff it is a κ reduction.

(iv) ι⊆κ

(v) if (f, f')∈κ then (f g, f' g)∈κ for each compositional morphism g

(vi) if ∀ (f, f')∈κ, (e g, e g')∈κ and h, h', f_h and g_h are morphisms fulfilling the points (a) to (d) from below then (h, h')∈κ (refer to Figure 15):

(e) $f = h\, f_h$ and $f' = h'\, f_h$,

(f) $g = h\, g_h$ and $g' = h'\, g_h$,

(g) e is the coequaliser of f and f'

∀ x, x': $N_h \to N_x$ holds x = y iff ($x\, f_h = y\, f_h$ and $x\, g_h = y\, f_h$)

Proof: If r is a κ reduction, it is clearly a reduction of the sub-relation ι. It remains to be shown that an ι reduction r equalises all morphisms related by κ. This is clear for (i). For (ii), if r equalises f and f' it also equalises fg and f'g. Because r equalises f and f' in case (iii), it factorises through e and hence ι also relates rg and rg', then g and g' are equalised by r and this implies that r also equalises h and h'. Hence r equalises all morphisms related by κ and, because all steps commute with left multiplication, r is a κ reduction♦ ⇨39⇦

7.5 From Structure to Behaviour and Semantics

7.5.1 Net Systems

Proposition 41. SYS0: *NET→*SYS forms a coreflection with NET. PP and F lift to systems yielding the commutative adjunction diagram of Figure 21.

Proof: Clearly [SYS0 N, S] ≅ [N, NET S] by interpreting the same morphism once in a system and once in a net is natural and injective. Because $f\,\mathbf{0} = \mathbf{0} \leq I_S$ it is also surjective and hence bijection as required by the claimed adjunction. The unit ID→NET SYS0 adds and removes an initial marking from the same net which is an isomorphism and turns the adjunction into a coreflection.

Let S' = (N', I') a place-preserving system. Interpret for a moment N' as an object of PPNET. The unit $\varepsilon_{N'}$: N'→NET PP N' gives a place-preserving morphism ε': N'→PP N'. Hence ε' I' is a marking in PP N' and PP S' = (PP N', ε' I') is well defined. Although ε' is not natural for ID→PP it is 'natural on places' which is sufficient for

$(PP\ f)\ I_{PP\ S} = (PP\ f)\ \varepsilon\ I_S = \varepsilon'\ f\ I_S \leq \varepsilon'\ I_{S'}$

hence PP is well-defined on morphisms.

Let $\eta_{N,N'}$: PTNET[NET N, N'] → PPNET[N, PP N'] the natural equivalence given by the adjunction from U and PP. η lifts to systems as $\eta_{N,N'}$: PTSYS[NET (N, I), (N',I')] → PPSYS[(N, I), (PP N', ε' I')] because of the same reason that ε' is natural on the initial marking I'.

F is lifted in the same way. Commutative adjunctions means that left adjoints are composed with left adjoints and right adjoints with right adjoints. That these compositions are commutative follows directly from the construction ♦ ⇨44⇦

Proposition 42. IM: *SYS→*SYS has a right adjoint, namely MM2I for PTSYS and PPSYS and FMM2I for FSYS.

Proof. We have to lift the functor MM2 and the appropriate part of Proposition 24 o systems. For that define the object function and counit ε_S by

MM2I S = (MM2 IP S, I)

$\varepsilon_S = \iota_S\ \varepsilon_{IP\ S}$: IM MM2I S → S with

$\varepsilon_{IP\ S}$: U PL MM2 IP S → IP S is the counit of the adjunction of U PL and MM2 and

ι_S: IP S→NET S the natural embedding.

This works fine on the initial marking and the universal property follows from that in PPNET. This proves the claim for PPSYS. It implies the claim for PTSYS because IM f is place-preserving for any place-transition morphism f: S→S' by

f_β supp I ⊆ supp I' ⊆ P' .

Finally, for FSYS we must define

FMM2I S = (F MM2 IP S, $\varepsilon_{\mathrm{MM2\ IP\ S}}$ I)
IM FMM2I S equals IM MM2I S thus the counit ε_S may be defined as above and F lifts the universality of MM2I to FSYS by Proposition 27. ♦ ⇨45⇦

Proposition 44. A diagram D in *SYS

- has a colimit iff IM D has a colimit j: IP D→J and the combined diagram NET (j: IM D→J, ι: IM D→D) has a colimit in *NET
- has a limit iff IM D and NET D have a limit

As prerequisite the universal construction for IM D only needs naturality, uniqueness may be dropped.

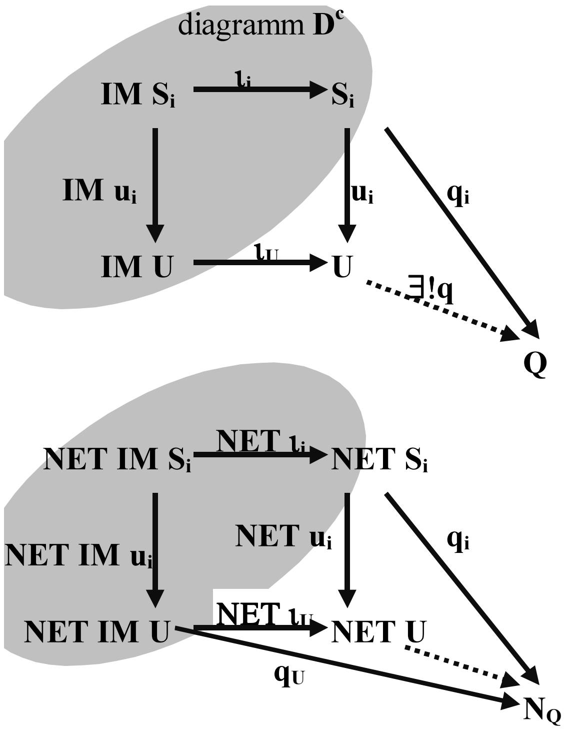


Figure 75. How IM and NET transfer the colimi U.

Proof: Let u: D→U the colimit of a diagram D in *SYS. By Proposition 42 IM has a right adjoint, preserves colimits and IM U is the colimit of IM D.

To show that NET U is the colimit in *NET let q = {q_C} a natural transformation from NET D^c to a net N_Q in *NET with D^c the combined diagram (IM u: IM D→IM U, ι: IM D→D) as shown in Figure 75. Then

Q = (N_Q , MAX {q_C I_C | C an object of D}

(with max the multiset maximum e.g. the maxima of each component) lifts N_Q to a system Q and q to a natural transformation (u: IM D→IM U, ι: IM D→D)→Q in *SYS. Thus by the universal property of U there is a unique connecting q_U in *SYS. NET q_U is - up to natural isomorphism - a natural connection from NET U to N_Q. Another such connection q' would lift back to the diagram in *SYS because commutativity implies the preservation of the initial marking. Thus, the universal property of U in *SYS implies q = q'. Hence NET U is the colimit of the diagram in *NET.

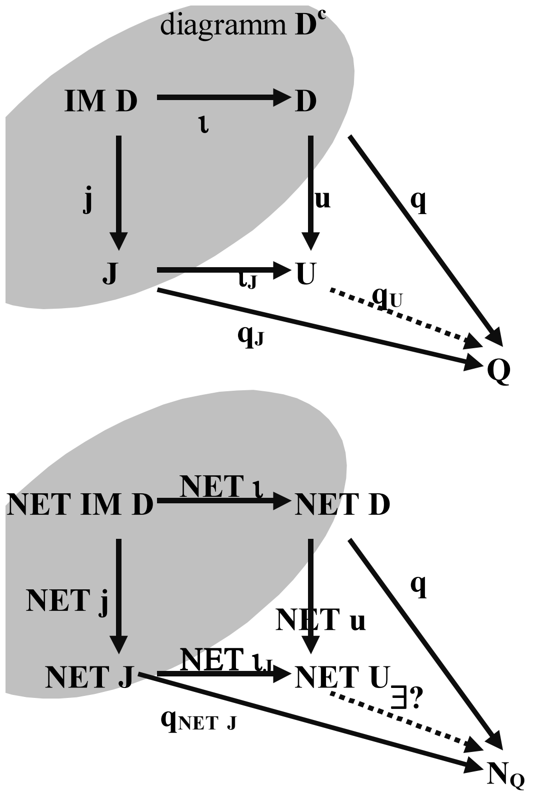


Figure 76. Construction of the colimit U.

For the if direction let j: IM D→J the colimit and N_U the colimit of the diagram NET D^c in *NET. As shown in Figure 76 D^c is this time the combined diagram (j: IM D→J, ι: IM D→D). The diagram is simplified over the previous one by using Diagram D instead of example object S_i. Now N_U is lifted to <u>*SYS</u> by

$U = (N_U, \iota_J\ I_J)$

again u preserves the initial marking by

$u_C\ I_C = u_C\ \iota_C\ I_{IM\ C} = \iota_J\ j_C\ I_{IM\ C} \leq \iota_J\ I_J = I_U$

To show that U has the claimed universal property let q a natural transformation D→Q in <u>*SYS</u>. Sending the whole diagram to NET yields a unique q: N_U→NET Q. It preserves the initial marking

$q_U\ I_U = q_U\ \iota_J\ I_J = q_J\ I_J \leq I_Q$

because q_J does. If q' is a connection from U to Q NET q' is a connection from N_U to NET Q. Hence by the universality of N_U and J follows q = q' and the universal property of U.

Given a J' that allows a natural connection that needs not to be unique define a subsystem J of J' as depicted by Figure 77:

$I_J = \mathrm{MAX}\ \{j_C'\ I_C \mid C \text{ an object of } D\}$

$X_J = \{p \in P_{J'} \mid I_J(p) \neq 0\}$

j': J→J' the obvious embedding.

J has still the property to factorise every natural transformation IM D→R. But additionally the connecting r is unique because every node of J is in the image of some j_C. Hence J is the colimit. The situation is shown in Figure 77. The bottom square is a simple push out square because NET j' is mono. u' is an embedding of N_U in N_U' with

$X_U' \setminus u'\ X_U = u_J'\ (X_{J'} \setminus (\mathrm{NET}\ j')\ X_J)$

N_U' is just N_U glued together with the extraneous nodes of J'. Hence the colimit N_U' exists exactly if the colimit N_U exists which implies by the above the colimit of the original diagram in <u>*SYS</u>. This proves the colimit case for the weakened prerequisite.

$$\begin{tikzcd}
\mathbf{IM\ D} \arrow[r, "\iota"] \arrow[d, "j"'] & \mathbf{D} \arrow[d, "u"] \\
\mathbf{J} \arrow[r, "\iota_J"] \arrow[d, "j'"'] & \mathbf{U} \arrow[d, dashed, "u'"] \\
\mathbf{J'} \arrow[r, "\iota_J'"] & \mathbf{U'}
\end{tikzcd}$$

Figure 77. The weakened precondition.

For the claim for limits let U the limit of D in <u>*SYS</u> as shown in Figure 78. A natural transformation r: R→IM D gives a natural transformation ι r : R→D and hence a unique r_U: R→U by the universality of U. But this r_U maps R to the initial marking of U hence it uniquely retracts to an $r_{IM\ U}$: R→IM U with ι_U r = r' and r is a unique factorisation of ρ through IM U. Thus IM U is the limit of IM D. Further NET U is the limit of NET D because NET has a left adjoint. Hence we got the only if direction for limits.

Reverse use Figure 79 with N_U the limit of NET D and J the limit of IM D. The diagram NET j: NET J→NET D is a natural transformation in <u>*NET</u> and yields by the universality of N_U a j_U: J→N_U. N_U may be lifted to <u>*SYS</u> by

$U = (N_U, j_U\ I_J)$

and

$u_C\ I_C = u_C\ j_U\ I_J = \iota\ j\ I_J \leq I_C$ for an object C of D

shows that u is a natural transformation in <u>*SYS</u>.

$$\begin{tikzcd}
\mathbf{R} \arrow[dr, dotted] \arrow[drr, dotted, "\exists\,!\ r_U"] \arrow[ddr, "r"'] & & & \mathbf{Q} \arrow[dl, dotted, "\exists\,!\ q_U"'] \arrow[ddl, "q"] \\
& \mathbf{IM\ U} \arrow[r, "\iota_U"] \arrow[d, "IM\ u"] & \mathbf{U} \arrow[d, "u"] & \\
& \mathbf{IM\ D} \arrow[r, "\iota"] & \mathbf{D} &
\end{tikzcd}$$

Figure 78. How IM and NET transfer a limit U.

Given a natural transformation q: Q→D in <u>*SYS</u> the universality of N_U yields a unique connecting q_U in <u>*NET</u>. IM q is a natural transformation IM Q→IM D yielding a connection q_J making the left triangle in *SYS in Figure 79 commutative. Sending q_J to *NET gives two connections NET (j_U q_J) and NET (q_U ι_Q) from NET Q to NET U $\cong$ N_U. By the universal property of N_U these two connections equal and hence the upper trapezoid in the diagram in *SYS also gets commutative. Finally

$$q_U\, I_Q = q_U\, \iota_q\, I_{IM\,Q} = j_u\, q_J'\, I_{IM\,Q} \leq j_U\, I_J = I_U$$

shows that q_U is a *SYS morphism and hence the claimed natural connection.

The uniqueness of q_U in *SYS follows from the uniqueness in *NET. Hence U is the claimed limit. Because no uniqueness property of J has been used the amendment is proved too ♦ ⇨45⇦

$$\begin{tikzcd}
IM\,Q \arrow[rrr, "\iota_Q"] \arrow[dr, dotted, "q_J"] \arrow[ddr, "IM\,q"'] & & & Q \arrow[dl, dotted, "\exists ? q_U"'] \arrow[ddl, "q"] \\
& J \arrow[r, dotted, "j_U"] \arrow[d, "j"] & U \arrow[d, "u"] & \\
& IM\,D \arrow[r, "\iota"] & D &
\end{tikzcd}$$

Figure 79. The construction of a limit.

7.5.2 Reachability and Liveness

Proposition 46. Unfolding the steps of a system to a state machine extends to a functor SM: PPSYS→SM.

Proof. Given a system S=(N, I) define SM S = (N', I')

- P' = all markings of S reachable from I including I itself
- T' = {(m, σ) | m∈ P' σ∈ 1S T such that the step σ is enabled at m'}
- pp(m, σ) = (m, m') with m' the follower marking from m under occurrence of σ, i.e. m[σ>m'.
- I' = I

Clearly this defines a state machine and it corresponds to the step unfolding of S. To define SM on morphisms let f: S→S', m[σ>m' a step occurrence in S and $\underline{\sigma} = \sum_{\substack{t \in T \text{ and} \\ f_B t \in T'}} \sigma(t)\ t$ then

SM f m = f m
SM f (m, σ) = if $\underline{\sigma}$ = 0 then f m else (fm, f $\underline{\sigma}$)

The first line is well defined because f is place-preserving. For the second line

$m\ [\sigma>$
$\Rightarrow m \geq \text{pre}\ \sigma \geq \text{pre}\ \underline{\sigma}$
$\Rightarrow f\ m \geq f\ \text{pre}\ \underline{\sigma} = \text{pre}\ f\ \underline{\sigma}$
$\Rightarrow fm\ [\underline{\sigma}>$

shows that f $\underline{\sigma}$ is enabled at fm and from m [σ> m' we derive

$f\ m' = f\ (m + (\text{post} - \text{pre})\ \sigma)$
$= f\ (m + (\text{post} - \text{pre})\ (\underline{\sigma} + (\sigma - \underline{\sigma})))$
$= f\ m + f\ (\text{post} - \text{pre})\ \underline{\sigma} + f\ (\text{post} - \text{pre})\ (\sigma - \underline{\sigma})$
$= f\ m + f\ (\text{post} - \text{pre})\ \underline{\sigma}$
$= f\ m + (\text{post}' - \text{pre}')\ f\ \underline{\sigma}$
$= (SM\ f)\ m + (\text{post}' - \text{pre}')\ (SM\ f)\ \sigma$

shows that SM f is a place-preserving morphism. f I = I' implies that SM maps the initial marking of S to that of S' and by induction over the length of step sequences in S that it maps reachable markings to reachable markings.

Hence SM f is a morphism from SM S→SM S'. Compositionality on markings follows directly from the definition and on transitions from the place-preserving property. This finishes the proof that SM is a functor ♦ ⇨46⇦

Proposition 48. SM: <u>PPSYS</u>→<u>SM</u> has neither a left nor a right adjoint.

Proof: Let N_{m-k} the net consisting of a single transition with m input places, k output places, all arc weights one and all m+k places disjoint. SM sends any N_{m-k} to N_{1-1}. Let ? a left adjoint to SM. What is $?N_{1-1}$? Consider $[?N_{1-1}, N_{3-1}] \cong [N_{1-1}, N_{1-1} = SM\ N_{3-1}]$. If the initial marking of $?N_{1-1}$ is zero there are either 1 or infinitely many morphisms in the left morphism set. Hence there must be at least one place with a single token. But the permutations of the three legs are automorphisms of N_{3-1}. Hence if there is not only the 0 morphism in the considered set there are at least 4 morphisms. Contradiction to a bijection between the two morphism sets.

For a right adjoint consider $[N_{1-1} = SM\ N_{3-1}, N_{1-1}] \cong [N_{3-1}, ?N_{1-1}]$. Again the initial marking cannot be zero and with the permutations of the input legs of N_{3-1} there are more morphisms in the right morphism set ♦ ⇨46⇦

Proposition 49. Let <u>*ND</u> the full subcategory of <u>*SYS</u> with neither dead transitions nor never marked places. The functor ND: <u>*SYS</u>→<u>*ND</u> removing dead transitions and never marked places forms a coreflection with the underlying functor.

Proof: Clearly ND is a functor. η: [U L, S] → [L, ND S] is just the interpretation of the same morphism once in <u>*ND</u> and once in <u>*SYS</u>. This is well defined because no transition in UL is dead and hence its image under a morphism is neither. Hence η is the required natural bijection and each unit ε_L: L→ND U L is obviously isomorphic yielding the claimed coreflection ♦ ⇨46⇦

7.5.3 Processes

Lemma 53. The image of a finite maximal place cut of a process is a reachable marking of the system. Any step sequence of a system is the image of a step sequence in an appropriate process.

Proof: Let r: R→S a process, C a maximal place cut of R and ρ_i = transitions of F^* C with depth i. Then

$I = C_0\ [\rho_1\!> C_1\ [\rho_2\!> C_2 \ldots [\rho_i\!> C_i \ldots$

is a step sequence of R with the C_i maximal place cuts. This is proved by induction over i.

Induction start i=0. I is a maximal place cut by the definition of a process.

Induction step from i-1 to i. σ_i is enabled because the input places of each transition in σ_i have a depth less than i and all these transitions occurred already and produced the tokens and because no place has more than one output transition the token is still there.

The occurrence of σ_i produces a marking C_i which is a place set by the properties of a process. We will show by contradiction that it is even a cut. If there were $x,y \in C_i$ with $x\ F^*\ y$ then x and y can not be both in C_{i-1} because this was a cut by induction hypotheses. They can not both be disjoint from C_{i-1} neither because then both had the same depth i which is incompatible with comparable. The last case $x \in C_i$ and $y \notin C_i$. Then $x\, F^*\, {}^\bullet y$ and there is at least one place $p \in {}^{\bullet\bullet}y$ with $x\, F^*\, p$ or $x = p$. The equality is not possible because ${}^\bullet y$ consumed the token in p. Neither is the first case because C_{i-1} was already a cut. Hence by contradiction C_i is a cut.

The same argumentation shows that C_i is maximal. Furthermore, because C is finite the maximal depth of elements of C is finite say k. $p \in C_k$ implies the transition ${}^\bullet p$ in some C_i but no transition in $p^\bullet$ in any C_j. This implies

$$p \in F^*\, C \text{ and } F^*\, C \cap p^\bullet = \{\}$$

and $p \in C$. Hence $C_k \subseteq C$ and by maximality $C_k = C$.

That r is folding and $r\, I_R \leq I_S$ implies that $r\rho_i$ is a step sequence in S. Together this is the first claim.

For the second claim build R and ρ_i by induction for a step sequence σ_i of S.

Induction start. R_0 has $|\, I_S \,|$ places with one initial token and no transitions. Choose a unitary r_0 such that $r_0\, P_0 = I_S$.

Step for i-1 to i. Add new transitions

$\tau_i = \{(t, i, k) \mid k \in [1, \sigma_i(t)]\}$

$T_i = T_{i-1} \cup \tau_i$

r_i an extension of r_{i-1} with $r_i\,(t, i, k) = t$

pre_i an extension of pre_i commutating with r_i and ${}^\bullet v$ are disjoint for all $v \in \tau_i$

$post_i$ an extension of $post_i$ that is commutating with r_i and $v^\bullet$ are disjoint from each other and P_{i-1}

Because σ_i is enabled it is possible in each step to find the necessary pre places of each new transition which makes the whole construction well defined and $r_i\ \tau_i = \sigma_i$ maps the step sequences ♦ ⇨48⇦

7.5.4 Weighted Occurrence Systems

Lemma 55. The image of the transitions of a process of a weighted occurrence system is a cut step and each cut step is such an image. Hence a marking is reachable iff it equals $(I - pre\,\gamma + post\,\gamma)$ for a cut step γ. A process of a weighted occurrence system of a system S yields a process of S.

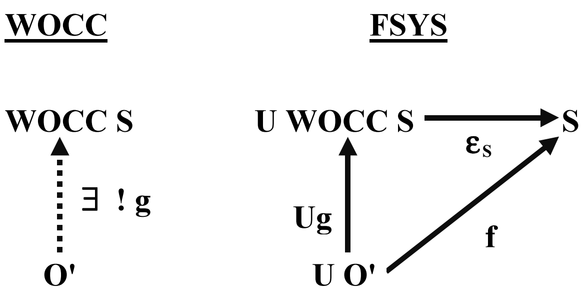
Proof: Let $r: R \rightarrow O$ a process of weighted occurrence system and σ_i the transition of T_R of depth i. Because R is a process

$$pre\ \rho_i \leq I_R + (post - pre)\ \sum_1^{i-1} \sigma_k$$

and because r is a folding

$$pre\ r\ \rho_i \leq I_O + (post - pre)\ \sum_1^{i-1} r\ \sigma_k$$

hence $r\ \rho_i$ is an enabled step sequence. If σ is a cut step



$$\begin{tikzcd} \text{WOCC } S \\ O' \arrow[u, dashed, "\exists\ !\ g"'] \end{tikzcd} \qquad \begin{tikzcd} U\ \text{WOCC } S \arrow[r, "\varepsilon_S"] & S \\ U\ O' \arrow[u, "Ug"] \arrow[ur, "f"'] & \end{tikzcd}$$

Figure 80. WOCC by an universal arrow from U to S.

$$\sigma_i = \sum_{t\in T \text{ with depth } i} \sigma(t)\, t$$

is an enabled step sequence because pre σ_i consists of places of depth less i that got marked in a previous step and because σ is a cut step got at least as many tokens as all follower transition in σ consume together. Hence the sequence σ_i is the image of a process of O.

Finally let o: O→S a weighted occurrence system of S and r: R→O a process of O. (o r): R→S is a process of S because o and r are unitary ♦ ⇨49⇦

Proposition 56. There is a functor WOCC: <u>FSYS</u>→<u>WOCC</u> which is right adjoint to the underlying functor U forming a coreflection..

Proof. As in the proof of Proposition 23, it is sufficient to unfold a system S into a weighted occurrence system WOCC S, to construct an universal arrow ε_S from U to S and to verify that the units are isomorphisms. The proof translates constructions from [MMS92] in our framework.

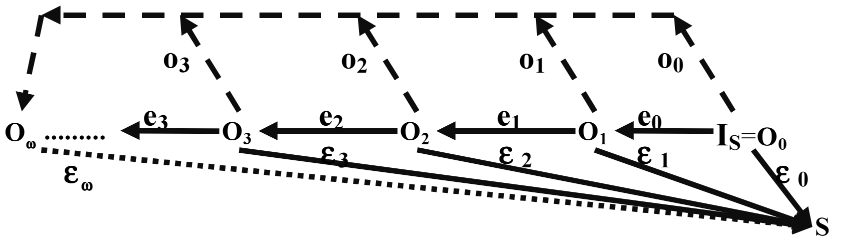


Figure 81. The infinite diagram and its colimit O_ω.

WOCC S is constructed as the colimit O_ω of the infinite diagram in Figure 81. There, O_ω is approximated by a sequence of weighted occurrence systems O_i of maximal depth i. The diagram is constructed by induction. Induction start: ε_0 is the natural embedding from O_0 = IM S into S.

Induction step from O_i to O_{i+1}. O_i is expanded as follows:

$T_{i+1} = T_i \cup \{(m, t) \mid t\in T_S$ is a transition of depth i+1 and
m is a reachable marking of O_i with $\varepsilon_i\, m = \text{pre } t\}$

$P_{i+1} = P_i \cup \{(m, t, p) \mid (m, t) \in T_{i+1}\setminus T_i \text{ and } p\in t^\bullet \}$

$pp_{i+1}(t') = \text{if } t'\in T_i \text{ then } pp_i\, t' \text{ else if } t' = (m, t) \text{ then } \left(m, \sum_{p\in t^\bullet} ((\text{post } t)(p))\,(m,t,p)\right)$

This defines an occurrence system $O_{i+1} = (pp_{i+1}, I_i)$ and the morphism e_i: $O_i \to O_{i+1}$ is the embedding used in the construction. ε_{i+1} is the unique extension of ε_i with

$\varepsilon_{i+1}(m, t) = t$ for $(m, t)\in T_{i+1}\setminus T_i$

$\varepsilon_{i+1}(m, t, p) = p$ for $(m, t, p)\in P_{i+1}\setminus P_i$

It is straight forward to see that with ε_i also ε_{i+1}: $O_{i+1}\to S$ is a weighted occurrence system of S and e_i is a monomorphism with $\varepsilon_i = \varepsilon_{i+1}\, e_i$.

This defines an infinite diagram D = (e_0, e_1, e_2, ...) and IM S is isomorphic to all initial markings hence by Proposition 44 the colimit o: D→O_ω exists. By the universal property of O_ω there is a unique ε_ω : $O_\omega\to S$ making the combined diagram commutative.

We claim that ε_ω : $O_\omega\to$ S is a weighted occurrence system of S. ε_ω and o_i are unitary because $\varepsilon_\omega\, o_i = \varepsilon_i$ which is unitary by construction. The o_i are monomorphic because X_ω is simply the union of the X_i. This implies that the preset of each place of P_ω consists of a single transition, that F_ω is acyclic, F_ω^* x is finite for any node x of X_ω and, thus, O_ω is an occurrence system.

Next we prove that $\varepsilon_\omega : O_\omega \rightarrow S$ is a universal arrow from U to S as depicted in Figure 80. Such an f: U $O' \rightarrow S$ may be expanded to the diagram in Figure 82. Its upper part redraws the construction of O_ω = WOCC S. In the lower part each O_i' is the subnet of U O' which consists of the nodes of depth at most i. The e_i' and ε_i' are the natural embeddings. O' equals the colimit O_ω' of the diagram (e_0', e_1', e_2', ...) and the monomorphisms $o_i' = \varepsilon_i'$: $O_i' \rightarrow O_\omega'$ make the diagram (e_0', ε_0', o_0', e_1', ε_1', o_1', e_2', ...) commutative.

By induction over i we show that there exist unique g_i which make the combined diagram (without g_ω) commutative. Induction start: g_0 must equal IM f. Step from i to i+1. If such a g_{i+1} exists it is defined on all nodes of depth less or equal i by $g_{i+1}\, e_i' = e_i\, g_i$ because e_i' is monomorphic. For any transition $t' \in T_{i+1}'$ of depth i+1 it must fulfil

$\quad$ pre g_{i+1} t' = g_{i+1} pre t' = $e_i\, g_i\, e_i'^{-1}$ pre t' and

$\quad$ $\varepsilon_{i+1}\, g_{i+1}\, t' = f\, \varepsilon_{i+1}'\, t'$.

This implies by the construction of O_{i+1}

$\quad$ $g_{i+1}\, t' = (f_\lambda\, \varepsilon_{i+1}'\, t')$ (m, t) with

$\quad\quad$ $t = f_\beta\, \varepsilon_{i+1}'\, t'$ and

$\quad\quad$ $m = (1\ /\ (f_\gamma\, \varepsilon_{i+1}'\, t'))\ g_i\ e_i'^{-1}$ pre t'

which uniquely determines the image of t'. For the postplaces of t' follows similarly:

$\quad$ $g_{i+1}\, p' = (f_\lambda\, \varepsilon_i'\, p')$ (m, t, $f_\beta\, \varepsilon_i'\, p'$) for $p' \in t'^\bullet$

Together, this uniquely determines the morphism $g_{i+1} : O_{i+1}' \rightarrow O_{i+1}$.

$$\begin{tikzcd}
O_\omega \arrow[rrrrr, bend left, "\varepsilon_\omega"] & \cdots & O_2 \arrow[l, "e_2"'] \arrow[rrr, bend left] & O_1 \arrow[l, "e_1"'] \arrow[rr, bend left] & I_S=O_0 \arrow[l, "e_0"'] \arrow[r, "\varepsilon_0"] & S \\
O'_\omega=O' \arrow[u, dotted, "g_\omega"] \arrow[rrrrr, bend right, "\varepsilon'_\omega=\mathrm{Id}"'] & \cdots & O_2' \arrow[l, "e_2'"'] \arrow[u, "g_2"'] \arrow[rrr, bend right] & O_1' \arrow[l, "e_1'"'] \arrow[u, "g_1"'] \arrow[rr, bend right] & I_O'=O_0' \arrow[l, "e_0'"'] \arrow[u, "g_0"'] \arrow[r, "\iota'_0"'] & U\ O' \arrow[u, "f"']
\end{tikzcd}$$

Figure 82. Universality of o_ω: $D_\omega \rightarrow O_\omega$.

To verify the existence of g_{i+1} notice that there is a step sequence σ' of U O' enabling t' by the definition of an occurrence system. $\varepsilon_i\, g_i\, \sigma'$ is a step sequence of S which enables $f\, \varepsilon_{i+1}'\, t'$ and thus t as defined above. Hence, (m, t) is indeed a transition of O_{i+1} and g_{i+1} is well defined by the above equations.

The $o_i g_i$ form a natural transformation from the diagram (e_0', e_1', e_2', ...) to O_ω. The universal property of O_ω' yields a unique connecting morphism g_ω: $O_\omega' \rightarrow O_\omega$ making the combined diagram commutative and $g = \varepsilon_\omega'^{-1}\, g_\omega$ is the required connecting morphism.

Thus, the existence of a connecting g is proved. For another g'' which factorises ε_ω, the image of o_i' g'' consists of nodes of depth less or equal i and o_i' g'' retracts to a g_i'': $O_i' \rightarrow O_i$. This yields again the diagram of Figure 82 for which we proved that the g_i are unique. Hence, g_i'' equals g_i and the universal property of O_ω' implies that g'' equals g.

Finally, a unit ε_O: WOCC U $O \rightarrow O$ of a weighted occurrence system is obtained in Figure 81 by setting S = U O. But as already mentioned, in this situation the O_i are simply the subnets of U O of nodes of depth less or equal i, and both U O and O_ω are the colimit of the diagram.

Hence, they are isomorphic and the unit $\varepsilon_\omega = \varepsilon_S$ is isomorphic. This finishes the proof that U and WOCC form a coreflection ♦ ⇨49⇦

7.5.5 Safe Occurrence Systems

Proposition 61. The underlying functor U: DEC→WOCC1 has a right adjoint DEC: WOCC1→DEC.

Proof. Proof: The proof is similar to that of Proposition 56. Let S be a weighted occurrence system of WOCC1. Again, DEC S is constructed as the colimit of the infinite diagram from Figure 81. O_0 is simply IM S. Induction step from O_i to O_{i+1}. To the transitions of O_i we add all combinations of presets of multiples of transitions of depth i+1 and decorations of output-places:

$$T_{i+1} = T_i \cup \{(m, \lambda t, \phi) \mid m\in 1S\, P_i,\ \lambda,\ \phi_m \in \mathbb{N}^+,\ t\in T_S,\ \sigma\colon [1, \phi_m] \to \mathbb{N}^+ t^\bullet \text{ with}$$
$$t \text{ has depth } i+1,\ \lambda t \text{ is not dead in S},\ \varepsilon_i\, m = \text{pre}\ \lambda t \text{ and } \sum_{k=1}^{\phi_m} \phi(k) = \text{post}(\lambda t)\}$$

$$P_{i+1} = P_i \cup \{((m, \lambda t, \phi), k) \in T_{i+1} \times \mathbb{N}^+ \mid k \le \phi_m\}.$$

pp_i and ε_i are extended by

$$pp_i\,(m, \lambda t, \phi) = \left(m, \quad \sum_{((m,\lambda t,\phi),k)\in P_{i+1}} ((m,\lambda t,\phi),k) \right)$$

$$\varepsilon_{i+1}\,(m, \lambda t, \phi) = \lambda t$$

$$\varepsilon_{i+1}\,((m, \lambda t, \phi), k) = \phi(k)$$

and e_i is the obvious embedding of O_i in O_{i+1}. Then the colimit o: D→O_ω exists. A decoration Φ on O_ω is defined by

$\Phi\ o_0\,(p) = 1$ and

$\Phi\ o_i\,((m, \lambda t, \phi), k) = k$ for $i > 0$

which turns DEC S = (O_ω, Φ) into a decorated occurrence system.

To show the universality of ε_ω: O_ω = U DEC S→S Figure 82 is reused. Induction start: IM g yields g_0 because I_S is a set. Induction step from g_i to g_{i+1}. If such a g_{i+1} exists it must fulfil for any transition t' of O_{i+1}' of depth i+1 and for all (b', k'), (b'', k'')$\in t'^\bullet \cap \text{def } f_\beta$

$\text{pre } g_{i+1}\, t' = g_{i+1} \text{ pre } t' = e_i\, g_i\, e_i'^{-1} \text{ pre } t'$,

$\varepsilon_{i+1}\, g_{i+1}\, t' = f\, \varepsilon_{i+1}'\, t'$

Furthermore, g_{i+1} must correspond to a decorated morphism, especially it must be binary. This yields by the construction of O_{i+1}

$g_{i+1}\, t' = (g_i\, e_i'^{-1} \text{ pre } t',\ f\, \varepsilon_{i+1}'\, t',\ \sigma)$ with

$\sigma : [1, |t^{\neq}|] \to \mathbb{N}^+ T'$ with $t^{\neq} = \{p'\in t'^\bullet \mid f\, \varepsilon_{i+1}\, p' \neq \mathbf{0}\}$,

$\sigma(k) = f\, \varepsilon_{i+1}\, p'$ for the $p'\in t^{\neq}$ with $k = |\{p''\in t^{\neq} \mid \Phi'\, p'' \le \Phi'\, p'\}|$

which uniquely determines the image of t' and its post-places. Thus, g_{i+1} is unique.

The existence of g_{i+1} follows from the facts that $g_i\, e_i'^{-1}$ pre t' is reachable in S because pre t' is reachable in O_i' and $\sigma([1, |t^{\neq}|])$ = post $f\, \varepsilon_{i+1}'\, t'$. It is easy to see, that the colimit of the ε_i yields the claimed morphism g. On the other hand, such a g' yields a similar diagram with g_i'. But, we have shown that g_i equals g_i' which implies the equality of g and g' ♦ ⇨51⇦

8 Appendix: Proofs and Details for Reverse Engineering

8.1 Introduction

8.2 Petri Nets as a Modelling Tool

8.2.1 Forward Engineering by Folding

Remark 64. Colouring of orders, shipments and boxes.

Define the colour morphism C: $N^{POSB} \to N^u$ by

C_β^{-1}(`allocating`) ≅ C_β^{-1} (`startA`) ≅ ORDER
C_β^{-1} (`shipping`) ≅ C_β^{-1} (`startS`) ≅ {(ord, shi) ∈ ORDER x SHIP
| shi is a shipment of ord}
C_β^{-1} (`packing`) ≅ C_β^{-1} (`startP`) ≅ {(os, bo) ∈ C_β^{-1} (`shipping`) x BOX
| bo is a box of os}
C_β^{-1} (`oPos`) ≅ {(old,ord) ∈ PART x ORDER | ord orders old}
C_β^{-1} (`allocate`) ≅ {(old, new, ord) ∈ PART x PART x ORDER
| (old, ord) ∈ C(`oPos`) and new is a valid replacement of old}
C_β^{-1} (`aPos`) ≅ {(new, ord) ∈ PART x ORDER | ∃ old with (old, new, ord) ∈ C(allocate)}
C_β^{-1} (`ship`) ≅ C_β^{-1} (`sPos`) ≅ {(pos, shi) ∈ C_β^{-1} (`aPos`) x SHIP | shi is a shipment of pos}
C_β^{-1} (`pack`) ≅ C_β^{-1} (`bPos`) ≅ {(pos, b) ∈ C_β^{-1} (`sPos`) x BOX | b is a box of pos}

The pre and post functions of the `startS` and `ship` transitions are as follows
pre(`startS`, ord, shi) = (`allocating`, ord)
post(`startS`, ord, shi) = (`shipping`, ord, shi)
pre(`ship`, par, ord, shi) = (`shipping`, ord, shi) + (`aPart`, par) + (`aPos`, par, ord)
post(`ship`, par, ord, shi) = (`shipping`, ord, shi) + (`sPart`, par) + (`sPos`, par, ord, shi)

Analogously the other transitions are modified. These modifications are straightforward, but, we ignore certain application logic such as:

- (`startS`, order, ship) must produce a unique combination of (order, ship)
- if all shipped positions (of an order) are packed, either `next` fires (iff the order has additional positions to allocate or ship) or `finish`.

Such application logic may be modelled by the non-deterministic firing of concurrently enabled transitions, by a marking dependent guard or by additional nodes ♦ ⇨58⇦.

Algorithm 65. The procedures `next` and `allocate`.

`next` checks whether all shipped parts are packed. If this is not the case then the FAIL statement rollbacks the database transaction:

```
PROCEDURE next (anOrder; aShip; aBox);
  BEGINTRANSACTION;
  IF EXISTS ( SELECT * FROM SPos WHERE order = anOrder) THEN FAIL ENDIF;
  DELETE packing WHERE order = anOrder AND ship = aShip AND box = aBox;
  INSERT INTO allocating (order) VALUES (anOrder);
  COMMIT;
END next
```

```
PROCEDURE allocate(oldPart; newPart; anOrder);
  BEGINTRANSACTION;
  IF NOT isReplacedByAvailablePart(oldPart, newPart) THEN FAIL ENDIF;
  LOCK allocating WHERE order = anOrder;
  UPDATE OPart SET (tokens := tokens - 1)
    WHERE part = oldPart AND tokens >= 1;
  UPDATE APart SET (tokens := tokens + 1) WHERE part = newPart;
  UPDATE OPos SET (tokens := tokens -1)
    WHERE order = anOrder AND part = oldPart AND tokens >= 1;
  IF EXISTS (SELECT * FROM APos
      WHERE order = anOrder AND part = newPart) THEN
    UPDATE APos SET (tokens := tokens + 1)
      WHERE order = anOrder AND part = newPart
  ELSE
    INSERT INTO APos (order, part, tokens) VALUES (anOrder, newPart, 1)
  ENDIF;
  COMMIT
END allocate;
```

`isReplacedByAvailablePart` is a boolean function that checks whether newPart is an available and legal replacement of oldPart, otherwise the `FAIL` statement will `ROLLBACK` the database transaction. The former function implements the check of the colouring set of `allocate` which contains major application logic i.e. the replacement rules.

The `LOCK` statement is an extension to SQL, to lock the allocating tuple for other transitions, as update statements implicitly do. The last IF statement implements an update insert switch. If an appropriate tuple of `aPos` already exists in the database the tokens attribute is increased otherwise a tuple is created.

Again the procedure corresponds to the input and output functions of the transition. The `LOCK` and the `UPDATE/INSERT` switch may either be considered as bridge between the two paradigms or as an optimisation. ♦ ⇨59⇦

Remark 66. Main Driver.

There are different options how to deal with the missing driver which decides when to fire which transition:

- Keep it missing: the Petri net model is abstracting from the driver.
- Write the driver as a Petri net simulator. I.e., the driver arbitrarily selects a procedure and executes it with arbitrarily selected parameters. If it fails then the corresponding coloured transition hasn't been enabled and the database rollback will cancel all effects of the failed procedure.
- Rewrite the procedures in order to win efficiency. For example `isReplacedByAvailablePart` would not test whether the input parameter `newPart` is a replacement but compute the appropriate value of `newPart`.
- Write the driver as a specialised Petri net interpreter using the concrete topology of the net. E.g., after firing of `ship`, `startP` could be fired for the same order and then `pack` for the part shipped before. Such an implementation could include additional business logic,

e.g. the rules to reduce the number of shipments and boxes without delaying emergency parts.
- As a variant, we could localise the specialised simulator and give every procedure the knowledge, which procedure/parameter combinations should be checked for transition enabledness.

♦ ⇨59⇦

8.3 Simple Reductions

Remark 67. Used Smalltalk constructs.

All coding fragments are formulated in Smalltalk. But, they are normally not a copy of the implementation in Smalltalk. They should illustrate the basic ideas. So often
- implementation details are skipped
- different methods are combined into one
- declarations or call arguments are not complete
- pseudo code or mathematical notation is used

The following is a cursory introduction to the pertinent Smalltalk constructs.

Comments are embedded in pairs of double quotes:
```
"this is a comment "
```
whereas strings are delimited by single apostrophes. In Smalltalk variables are references to objects. An assignment
```
var := 'abcd'.
```
assigns a reference to a string object to the variable var. Dots are used to separate statements.

Method headers for methods without arguments take the form
```
ClassName>>unaryMessage
ClassName class>>unaryMessage
```
The first is an instance method the second a class method. By convention Class names and global variables start with an uppercase character other variable and method names with a lowercase character. Embedded uppercase characters are used to enhance readability for names consisting of multiple words. Keyword messages use a varying number of keywords terminated by colons, the names of the formal arguments are declared after the colons:
```
ClassName>>messageNameWith: argumentOne andAnother: argumentTwo
```
The method name above is `messageNameWith:andAnother:` the first formal argument is called `argumentOne`, the second `argumentTwo`. After the header follows the declaration of local variables enclosed between vertical bars:
```
| localVariable anotherLocalVariable |
```
The basic Smalltalk construct is sending a message to a receiver with arguments. The format depends on the message type. For unary messages it looks like:
```
Array new.
self size
```
On the first line the message *new* is sent to the class Array. On the second line the message size is sent to the pseudo variable self which designates the current receiver. Binary messages consist of one or two special characters:
```
3 + 4
x == y
```
The first example sends the message + to the object 3 with an argument of 4. The second sends the message == to the object x with argument y. The priority of message sending is

- expressions within parentheses
- primary messages
- binary messages
- keyword messages
- within the same message class strictly from left to right (contrary to usual mathematical notation!)

example:

```
  newEmbs add: (self
     embeddingAtNodes: emb image ∪ nd withNeighbours
     ifAbsent: [self combine: emb newType:
          (self embeddingAtTrans: nd)]).
```

The evaluation of this expression is depicted in Figure 83. Message names are shown on a grey background. From top to bottom the message are in the temporal order of message sending. There is one exeception, namely the expression in square brackets which is a block. It is passed as a code fragment to the method `embeddingAtNodes:ifAbsent:` which may evaluate it an arbitrary number of times.

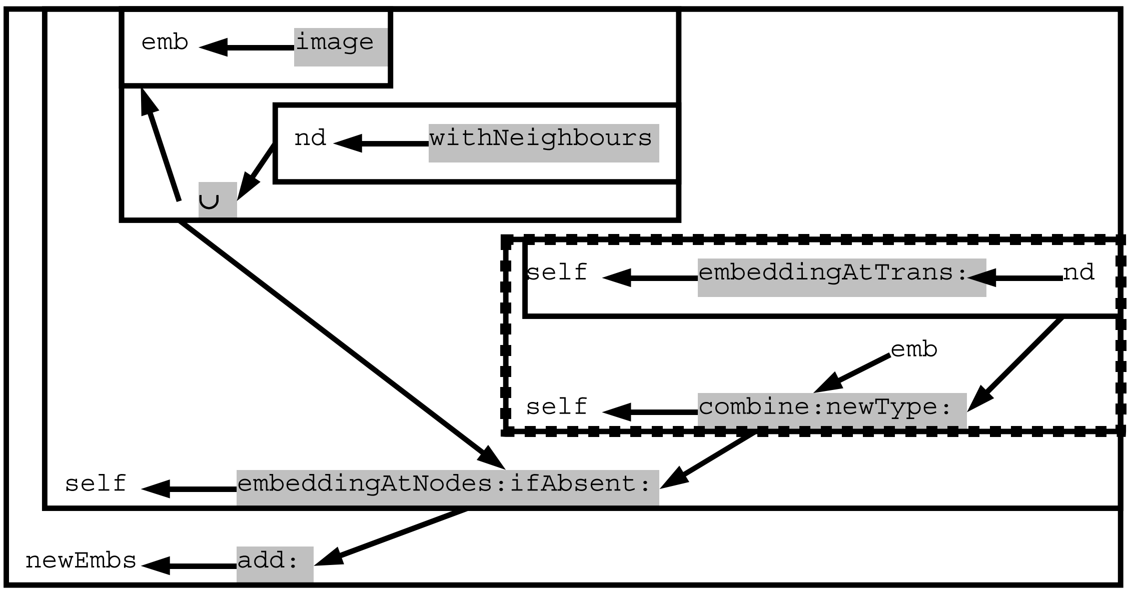


Figure 83. The evaluation of a nested Smalltalk expression.

Messages can be cascaded by semicolons:

```
  x root add: 1; add: (1+1); add: 3
```

A message after a semicolon is sent to the same receiver as the previous message. The message sequence of the example is depicted in Figure 84.

Each invocation of a method returns a result. This is coded as

```
  ^ x + y
```

to return the result of x + y. If a method execution reaches the end of a method it returns `self`. The ^ jumps out of method that contains the ^ . Execution of a ^ in a block which is passed as an argument to a method may terminate a whole cascade of method invocations.

A code fragment called a block is coded in square brackets possibly with arguments and local variables:

```
   [x]
   [:argument | argument doit]
   [:argument | | var1 | ....]
```

A block can refer to variables of the declaring procedure which supports the powerful feature of lexical closure. To evaluate a block send it the message **value** or **value:** or **value:value:** etc. depending of the number of arguments. It returns the value of the last expression and an empty block returns `nil`.

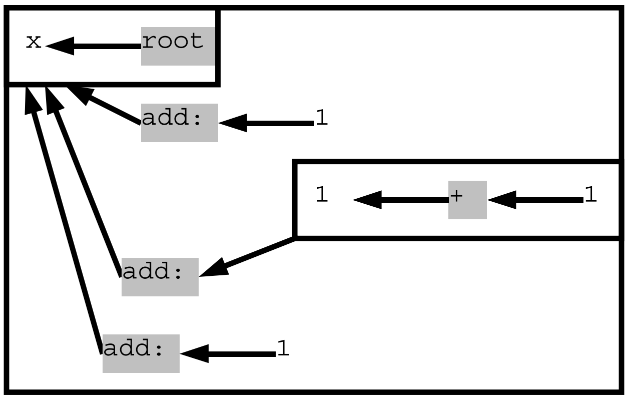


Figure 84. The evaluation of a cascaded Smalltalk expression.

The pseudo variable `self` holding the receiver has already been mentioned. Also `super` designates the receiver but method search starts only from the superclass of the method containing super. There are three further pseudo variables designating unique objects (singletons):

- `nil`: the uninitialised object
- `false`: the boolean value false
- `true`: the boolean value true

The message `yourself` simply returns the receiver. This is useful at the end of a cascade of messages: it returns the receiver instead of the result of the last message.

Flow control is implemented by messages and blocks:

```
  x < y ifTrue: ['less'] ifFalse: ['otherwise']
```

sends the keyword message `ifTrue:ifFalse:` to the boolean result of the comparison. Sent to true the first block is evaluated, sent to false the second. As the example shows this flow control construct may also be used for conditional expressions. A loop is coded as

```
   [ix * ix < 100] whileTrue: [ix := ix + 1]
```

The message `whileTrue:` is sent to a block. If it evaluates to true the second block is evaluated and loop cycles back by evaluating again the first block. There are some variants of these messages – the names are self-explaining.

Smalltalk offers a rich variety of collection classes. The basic iteration message is do:

```
   (5 to: 10) do: [:number | sum := sum + number]
```

`5 to:10` creates a collection of the integers from 5 to 10. The `do:` method evaluates the passed blocked for each element of the collection, passing the element into the block by the argument. Arrays and Dictionaries use the `at:` message to retrieve an element by a key. E.g.

```
  dict at: 'eins'
```

returns the object stored under the key 'eins'. If the collection does not contain the key the message fails badly. To avoid a failure code

```
  dict at: 'eins' ifAbsent: [do something]
```

If the key is found the corresponding object is returned otherwise the result of evaluating the block. There are useful variants like

```
  dict at: 'eins' ifPresent: [:object |
     object doSomethingObjectIsPresent].
```

```
   dict at: 'eins' ifAbsentPut: [newObjectThatWillBeAddedForKey].
```
The first example evaluates the block if the key is present with the object passed as an argument otherwise returns nil. The second adds an object if the key is missing. The `add:` message is used to add an object to collections without keys for example

```
   (OrderedCollection new add: 2; add: 3: add: 5; yourself)
```
This creates a new instance of `OrderedCollection`, adds 3 prime numbers and returns the newly created collection. The yourself is really necessary because probably for historic reasons `add:` returns the added argument not the receiving collection.

It is Smalltalk tradition to use very long names – if necessary half sentences – to make programs as readable as possible. So we stop the description of methods here and hope that the reader is able to guess the function of the methods not covered in this short introduction ♦ ⇨66⇦

8.3.1 Seed Types

Lemma 70. The seed types may be computed with cost O(e) with e the number of arcs of the net N.

Proof: The critical points for this bound are

- an efficient hash table, that limits the cost of an access with a key of length k to O(k).
- the representation of the automorphisms of a type. In our implementation we represent every automorphism as table of origin nodes to destination nodes. This has a worst case complexity of O(k k!) for a transition with k arcs. We use the small transition precondition of Definition 69, that k is a globally limited small number. If this condition would not hold, one could use the trace classes to give a more compact representation of the automorphism group. This is easy here, but will become less harmless for iterations over automorphisms ♦ ⇨68⇦

8.3.2 Combining Types

Algorithm 73. The combination of two types.

If we combine a subnet we get the situation shown in Figure 85. `connA` and `connB` are partial maps making the two triangles commutative.

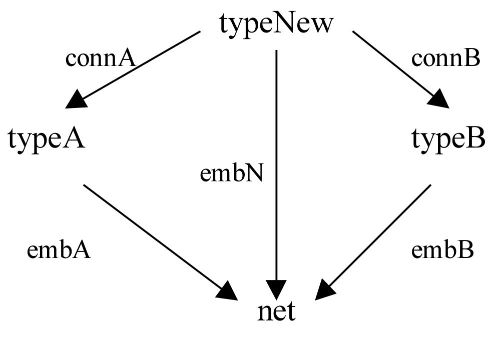

$$\begin{tikzcd} & \text{typeNew} \arrow[dl, "\text{connA}"'] \arrow[dr, "\text{connB}"] \arrow[dd, "\text{embN}"] & \\ \text{typeA} \arrow[dr, "\text{embA}"'] & & \text{typeB} \arrow[dl, "\text{embB}"] \\ & \text{net} & \end{tikzcd}$$

Figure 85. The combination of two types.

The computation of the automorphisms and embeddings of typeNew checks every pair of embeddings from `typeA` and `typeB`. For every such pair with every pair of automorphisms from `typeA` and `typeB` respectively it adds the merger of `newA°autoA°connA` and `newB°autoB°connB` to the collection `embs`, if they are not conflicting. The test for conflicts uses a little optimisation: it tests only on the common origins of `connA` and `connB`: This is sufficient, because all involved maps are injective and the union of the images has the correct size. But obviously the following code fragment is written to clarify the basic ideas, not for efficiency:

```
NetStructure>>createAutomorphismsEmbeddings.......
   .............
common := (connA origin ∩ connB origin) asArray.
typeA embeddings do: [:newA |
  typeB embeddings do: [:newB |
    (union := newA image ∪ newB image) size = embN size ifTrue: [
      embs := OrderedCollection new.
      typeA automorphisms do: [:autoA |
        typeB automorphisms do: [:autoB |
          (connA atAll: (autoA atAll: common))
              = (connB atAll: (autoB atAll: common)) ifTrue: [
            embs add: ((newA ° autoA ° connA)
                ∪ (newB °autoB ° connB))]]].
      embs isEmpty ifFalse: [(union isSameSet: embN image)
        ifTrue: [typeNew automorphisms addAll: embN⁻¹ ° embs]
        ifFalse: [self embeddingAtNodes: union ifAbsent: [
          self embeddingAdd: embs first]]]]
```

Efficiency cannot be too good for this algorithm because it allows to compute all automorphisms of `typeNew` and applied iteratively all automorphisms of the net, which is a NP-hard (subgraph isomorphism in [Gar79]) problem. So we try to avoid creating big types ♦ ⇨68⇦

Algorithm 74. Expanding problematic types.

The problematic types are identified by the argument that contains a collection of embeddings. The source of each embedding is the type and the image the location in the net at which the conflict occurred.

```
NetStructure>>newLiTypesExpandingAll: embeddingColl
  "expand each type embedding in embeddingColl
    until there are new types created"

  | oldSize oldEmbs newEmbs |

  newEmbs := embeddingColl.
  oldSize := self types size.
  [self types size = oldSize] whileTrue: [
    oldEmbs := newEmbs.
    newEmbs := OrderedCollection new.
    oldEmbs do: [:emb |
      emb image graphNeighbours do: [:nd |
        nd isTransition ifTrue: [newEmbs add: (self
          embeddingAtNodes: (emb image ∪ nd withNeighbours)
          ifAbsent: [(self
            combine: emb
            newType: (self embeddingAtTrans: nd))])]]]].
```

This method expands the embeddings by one transition, until new types are created and added to the net structure ♦ ⇨69⇦

8.3.3 Classifying the Nodes

Algorithm 75. Classification by type membership.

The cost of this algorithm as coded at ⇨69⇦ increases with the number of automorphisms. But this can be corrected by the use of trace classes of nodes (as discussed in 4.3.1 Seed Types):

```
NetStructure>>membershipForTypes  "optimised version"
   ...
   self types do: [:type |
     type embeddings do: [:emb |
       type trace equivalenceClasses do: [:eqCla |
         eqCla elements do: [:node | (byNode
           at: (emb at: node)
           ifAbsentPut: [self newSet]) add: eqClass]]]]
....
```

`byNode` now maps a node to an equivalence class of type nodes instead of collection of nodes. The union of the elements of the equivalence classes equals the collection formerly computed. The final effect is hence the same but automorphisms decrease complexity instead of increasing it ♦ ⇨69⇦

Algorithm 76. Convert membership classification to an equivalence relation.

```
NetStructure>>equivalenceForMembership: membership
   | first relation |

   relation := self newEquivalenceRelation.
   first := Dictionary new.
   membership keysAndValuesDo: [:nod :mbrId |
     relation relate: nod to: (first at: mbrId ifAbsentPut: [nod])].
   first := nil. "relate all nodes, not contained in membership"
   self net nodes do: [:nd | relation
     equivalenceClassAt: nd
     ifAbsent: [first isNil
       ifTrue: [first := nd]
       ifFalse: [relation relate: first to: nd]]].
^ relation
```

This method translates the classification received as an argument called `membership` to an equivalence relation. Additionally it relates all nodes of the net that are missing in membership ♦ ⇨70⇦

8.4 Neighbourhood Reductions

8.4.1 Algorithm

Algorithm 83. Merging Maps.

The code looks more complicated than it really is. Figure 45 shows the situation including the names used in the following Smalltalk code.

```
Transition>>mergeMapsTo: aTrans info: info
| res map myEmb othEmb fixPoints myNd othNd myRed othRed |

  res := OrderedCollection new.
  myEmb := info structure embeddingAtTrans: self.
  othEmb := info structure embeddingAtTrans: aTrans.

  myEmb source automorphisms
    do: [:auto |
      map := IdentityDictionary new.
      fixPoints := self newSet.
      ((myEmb source places conform: [:tyNd |
        map
          at: (myNd := myEmb at: (auto at: tyNd))
          put: (othNd := othEmb at: tyNd).
        (myRed := info reduce at: myNd) = (info reduce at: othNd)
          ifTrue: [fixPoints add: myRed].
        info relation is: myNd equivalent: othNd])
      and: [fixPoints notEmpty]) ifTrue: [
        res add: map]].
^ res
```

The method computes the fixpoints for each automorphism of the receiver transition. If there are fixpoints the map is added to the result ♦ ⇨76⇦

Algorithm 84. Adapting to different adjacency criteria.

`Transition>>mergeMapsTo:info:` is the only method that needs to be modified to implement the different adjacency definitions. This is already prepared there because the whole fixpoint set is computed which is not necessary for the existence of fixpoints alone. The following lines of Smalltalk must be added before the loop:

```
  redInt := (search reduce atAll: (myEmb image)) ∩
            (search reduce atAll: (othEmb image)).
```

`redInt` contains the intersection of the image of the receiver and argument transition in the reduction. Then the variations of Definition 37 are implemented as follows:

- overlap: `redInt notEmpty`
- common fix point: `fixpoints notEmpty` (as coded)
- common intersection: `redInt isSameSetAs: fixpoints`
- clean intersection the last expression in the `conform:` block must be replaced by:

```
  ((redInt includes: myRed) or: [redInt includes: (info reduce at:
othNd)])
    ifTrue: [myRed = (info reduce at: othNd)]
    ifFalse: [info relation is: myNd equivalent: othNd]]) ♦ ⇨76⇦
```

8.4.2 Analysing the Spare-Part System

8.4.3 Complexity

Proposition 87. The reduction algorithm has in the worst-case cost of

O(e log (min (maxDeg(N'), | P |) γ)) with

- e the number of edges in the source net,
- maxDeg(N') the maximum number of arcs incident to a place of the reduced net,
- log the binary logarithm and
- γ a slowly increasing inverse of the Ackermann function, which does not surpass 3 in any real case

Proof. Let's first count how many times each message is sent. A place is added to the `toDo` list, whenever two places are merged, thus, maximally |P| times. If it is used also for initialisation the number doubles in the worst case. For each place in `toDo` `PlaceReduction>>mergeArcsInfo` is called once.

`PlaceReduction>>merge:` is only called after two places are merged thus maximally |P| times. `Place>>merge:info:` is called for every merged transition, for every neighbour, for every automorphism. Let maxAuto the maximal number of automorphisms of a single transition type. With this the limit gets maxAuto by the number of arcs e. Again we remark that this is acceptable by the small transition property (Definition 69).

We designate the number of runs of `PlaceReduction>>mergeArc:at:info:` with α and will calculate them in a moment. The remaining two methods run for each α for each arc at the same key, which is limited by the small constant Γ from the small transition property. So we get the following table:

method execution	**worst case count**
`toDo add:`	2 |P|
`Place>>merge:info:`	maxAuto e
`PlaceReduction>>merge:`	|P|
`PlaceReduction>> mergeArcsInfo:`	2 |P|
`PlaceReduction>>mergeArc:at:info:`	α ≤ e log(4 min(maxDeg(N'), |P|))
`Transition>>mergeMapsTo:info:`	α Γ
`Transition>>tryToMerge:info:`	α Γ

e is the number of edges of the source net and maxDeg(N') the maximum number of edges of a place in the reduction. With this notation the following three lemmas will be proved:

Proposition 87 / Lemma (i). The invocation count α is limited by e log(4 min(maxDeg(N'), |P|)). ⇨140◈

Proposition 87 / Lemma (ii). The total cost of the algorithm excluding the operations on equivalence relations is limited by O(α). ⇨141◈

Proposition 87 / Lemma (iii). The operations on equivalence relations cost O(α γ). ⇨142◈

Together this proves the proposition ♦ ⇨78⇦

Proposition 87 / Lemma (i). The invocation count α is limited by e log(4 min(maxDeg(N'), |P|)).

Proof. The invocations of `PlaceReduction>>mergeArc:at:info` can be partitioned in three groups:

- During initialisation we get e invocations, to copy each arc. This results in an initial set of place reductions with a totally $e_1 \leq e$ edges.
- The latter invocation either merge the new arc with an existing, which cannot happen more than e_1 times
- or add the remaining β times the new arc to the receiver.

Let

- r be an instance of `PlaceReduction` at a certain point in the algorithm
- e(r) be the number of edges already merged, i.e. `arcsByKey size`
- $e_1(r)$ be the sum of the number of edges from initial `placeReductions` that have been merged until now into r
- origs(r) the places of the original net that have been merged until now into r
- o(r) be the cardinality of origs(r) which equals the number of initial `placeReductions` that have been merged until now into r
- $\beta(r)$ the accumulated number of β-runs, moving the $e_1(r)$ initial edges to the e(r) current edges.

At the end of the algorithm we obviously get β as the sum of the $\beta(r)$ over each r corresponding to a place in the reduced net. Because $\alpha \leq \beta + 2e$ and $e_1 \leq e$ it suffices to prove

- $\beta(r) \leq e_1(r) \log(o(r))$
- $\beta(r) \leq e_1(r) \log(e(r))$

This proved by induction over $\beta(r)$ starting with $\beta(r) = 0$. This corresponds to the moment in the algorithm immediately after the initialisation, when the arcs of the source nets are copied in the arc maps of the place reductions. There is still one place reduction for each place in the original net. In this situation $o(r) = 1$, $e(r) = e_1(r)$, $\beta(r) = 0$ thus the start is ok.

Step: Let the last invocation of `PlaceReduction>>keysAndArcsTreeDo:` merge r' and r'' into r with

- $e' = e(r') \geq e'' = e(r'')$
- $e' \leq e = e(r) \leq e' + e''$
- $e_1' = e_1(r')$, $e_1'' = e_1(r'')$,
- $e_1 = e_1(r) = e_1' + e_2'$
- $o(r) = o(r') + o(r'')$
- $\beta(r) = e - e' + \beta(r') + \beta(r'')$

For the first limit we get:

$$\begin{aligned} \beta(r) &= e - e' + \beta(r') + \beta(r'') \\ &\leq e'' + e_1' \log(o(r')) + e_1'' \log(o(r'')) \\ &\leq e_1' (1 + \log(o(r'))) + e_1'' \log(o(r'')) && \text{because } e'' \leq e' \leq e_1' \end{aligned}$$

$\leq e_1' \log(o(r)) + e_1'' \log(o(r))$ case (i) $o(r') \leq o(r'')$
$= e_1 \log(o(r))$

In the case (ii) o(r') is greater than o(r'') and the 1+ in the middle line has to be moved to the right side term. Similarly we proof the third limit:

$$\begin{aligned}
\beta(r) &= e - e' + \beta(r') + \beta(r'') \\
&\leq e'' + e_1' \log(e') + e_1'' \log(e'') \\
&= e - e' + e_1' \log(e) - e_1' \log(e/e') + e_1'' \log(e'') \\
&\leq e_1 \log(e) + (e - e') - e' \log(e / e') && \text{because } e' \leq e_1' \\
&\leq e_1 \log(e) + (e - e') - e' \log(1 + (e - e') / e') \\
&\leq e_1 \log(e) + (e - e') - e' ((e - e') / e') \\
&= e_1 \log(e)
\end{aligned}$$

In the first step we used, that e and e_1 coincide for initial place reductions and in the second last step the inequality for the binary logarithm shown in Figure 86.

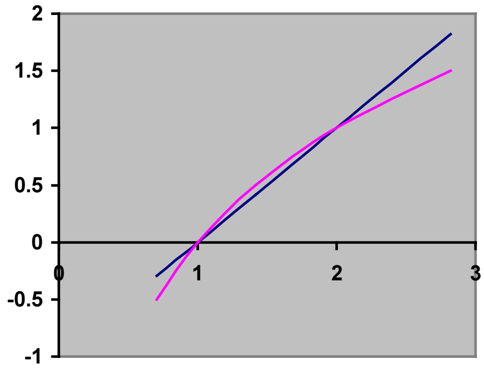


Figure 86. The inequality log(1+ x) ≥x in the interval [1,2].

The application of this lemma needs some care because the edges counted by e(r) and $e_1(r)$ do not exactly correspond to real edges in the source or reduced net. The arcs in a place reduction correspond really to trace classes of transition types. Thus edges belonging to the same trace class are merged, but, if edges belonging to different classes are merged then the map still contains them in each class. According to the small transition assumption we can ignore this for performance considerations.

The second limit may by improved by simply subtracting the value computed for o(r) = 1:

$$\beta(r) \leq e_1(r) \log(e(r)) - \sum_{q \in origs(r)} e_1(q) \log(e(q))$$

$$= \sum_{q \in origs(r)} e(q) \left(\log(e(p)) - \log(e(q))\right)$$

$$= \sum_{q \in origs(r)} e(q) \log\left(\frac{e(r)}{e(q)}\right)$$ ♦ ⇨139⇦

Proposition 87 / Lemma (ii). The total cost of the algorithm excluding the operations on equivalence relations is limited by O(α).

Proof. Obviously, we have to show that the net cost of each method invocation is limited by a constant. However,. charging each statement to its method does not work because of the loops. But, in most loops there are method calls that are counted in α. If all the statements in a loop and the control logic of the loop are charged to a method invocation within this loop then our goal is reached. This works fine except for `Transition>>mergeMapsTo:info:`. But here, we can use the constant Γ from the small transition property.

But is the cost of all these statements limited by a constant? There is one area to check: sets and maps. Both are implemented by hash tables. So, a single access or update at a key has constant cost if the overhead of the re-hashing is distributed to the updates.

A prerequisite is that the hash function generates a well behaving distribution. This is normally taken for granted, but, it might get tricky. The reason is that three different requirements have to get balanced:

- equal objects must get the same hash
- uniform distribution
- fast computation

For example it is not obvious how to implement that for a set with equality. One implementation used bit buckets and mapped thousands of different sets to hash 0 which forced a total redesign. ♦ ⇨139⇦

Proposition 87 / Lemma (iii). The operations on equivalence relations cost O(α γ).

Proof: The algorithm uses equivalence relations dynamically. I.e., the merging of equivalence classes and the access to the equivalence classes are done interleaving. The requirement to fulfil is that a single access and a single merge are done in constant time. The problem is that the merger of two classes needs to redirect all elements of one class to the other. The alternative is linking the classes. With this approach the access paths get longer and longer. Even with accountancy tricks to redistribute the costs they don't fall below O(log(n)).

The solution we present here combines all three techniques:

- linking
- redirecting
- accountancy.

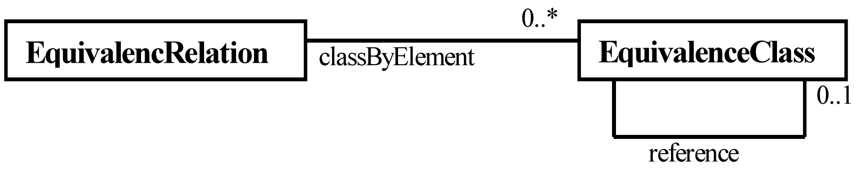


Figure 87. The class diagram for an equivalence relation.

EquivalenceRelation contains the map from element to class, and delegates linking and redirecting to the equivalence classes:

```
EquivalenceRelation>>equivalenceClassAt: anElement
  ^ (self classByElement at: anElement) finalReference
```

finalReference follows the links and on the way redirects all links:

```
EquivalenceClass>>finalReference
  ^ self reference isNil
     ifTrue: [self]
     ifFalse: [self
        reference: self reference finalReference;
        reference]
```

To relate two elements should neither create cyclic references nor overwrite existing references:

```
EquivalenceRelation>>relate: firstElement to: secondElement
  (self classByElement at: firstElement)
    swallow: (self classByElement at: secondElement)

EquivalenceClass>>swallow: secondClass
  self finalReference swallowFinal: secondClass finalReference

EquivalenceClass>>swallowFinal: secondClass
  self == secondClass ifTrue: [self].
  secondClass reference: self
```

This implements the basic functionality of equivalence relations. The code shows some add-ons such as lazy creation of absent equivalence classes. If the elements of the equivalence class are also needed a tree structure like the one shown for `PlaceReduction>> mergeArcsInfo` in Algorithm 79 may be used.

The computation of the cost of this algorithm is not easy. The problem is known from the literature as Union-Find and the algorithm as quick merge with path compression. Under the precondition that the `secondClass` has less elements than the receiver [Tar75] (cited in [Lee90] or [Meh86]) proved an upper limit for the cost of

$O(s + f\ \alpha(f, s+1))$ for s invocations of `swallow:` and $f > s$ invocations of `finalReference`.

Here, α is a slowly growing inverse of a variant of the Ackermann function:

α, A	$: \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}^+$	
$\alpha(f, s)$	$= \min\{z \in \mathbb{N}^+ \mid A(z, 4\lceil f/s \rceil) > \log s\}$	
$A(0, x)$	$= 2x$	$\forall x \geq 0$
$A(i, 0)$	$= 1$	$\forall i \geq 1$
$A(i+1, x+1)$	$= A(i, A(i+1, x))$	$\forall x \geq 0$

$\alpha(m, n)$ is less or equal to 3 for $\log(n) < A(3, 4)$ which is already a really astronomical figure (2^{16} nested powers of 2).

There are other approaches in the literature to deal with the cost of this algorithm. [Knu78] considers the sequence of finds and unions as a random process and computes the average cost. Even for the slower algorithm which for each union merges the sets, the cost is linear in the number of finds and unions. However, this depends on the definition of the random process. Our algorithm produces a random process somewhere between the faster and slower variant discussed in [Knu78].

[Gut98] gives an algorithm with cost linear in the number of finds if the unions are done compatible to certain classes of graphs. Again these limits do not directly apply to our algorithm. The paper mentions a lower bound of $O(m\ \alpha(f, s+1))$ for pointer machines. But for a RAM machine the paper gives linear cost for restricted classes of graphs and conjectures linearity for the general case.

The first result settles linearity for all practical cases and the further results show the possibility for linearity in general, hence, we do not investigate this any further ♦ ⇨139⇦

8.5 Integrating Domain Heuristics

8.5.1 Reduction Refinement

Algorithm 88. Relationships from Types.

```
NetStructure>>relationshipsFromTypes
  self relationships: Dictionary new.
  self types do: [:typ |
    typ embeddings do: [:emb |
      typ places do: [:from |
        typ places do: [:to | from = to
          ifFalse: [self
            relationshipOrigin: (emb at: from)
            image: (emb at: to)
            path: (Array
              with: (typ traceEquivalence equivalenceClassAt: from)
              with: (typ traceEquivalence equivalenceClassAt: to)
                )]]]]].
```

The used method `relationshipOrigin:image:path:` adds a tuple to the relation in the table corresponding to the path with the usual lazy initialisation trick:

```
NetStructure>>relationshipOrigin: from image: to path: path
  (self relationshipAt: path ifAbsentPut: [
      (self newRelationship source: path first;
        dest: path last; yourself)])
    origin: from image: to
```

The last technicality is to bother about the direction of paths. As we deal with undirected paths a path and its reverse have to match:

```
NetStructure>>relationshipAt: path ifAbsentPut: aBlock
  ^ self relationships
      at: path
      ifAbsentPut: [
        self relationships
          at: path reverse
          ifPresent: [: rel | ^ rel inverse ].
        aBlock value]
```

In fact, this contains another technicality...: The inverse of a relationship is not a copy, but a dynamic extension of the original relation such that an update of the original or the inverse is automatically propagated to the other relation. ♦ ⇨80⇦

Algorithm 91. Memberships compatible with sidedness.

```
NetStructure>>membershipAddRelationshipsCompatibleOneSide: mShip
  self relationships do: [:rel  |
    rel oneSide isNil ifFalse: [
      self membershipAddRelationship:rel compatibleOneSide: mShip]]

NetStructure>> membershipAddRelationship: rel compatibleOneSide: mShip
  | mark mbrVal|

  rel oneSide do: [:one |
    mbrVal := mShip ocAt: one.
    (rel allAt: one) do: [:at1 |
      ((mShip at: at1) isSameSet: mbrVal) ifFalse: [^ self]]].
  rel pairs keysAndValuesDo: [:from :to |
    ((mShip at: from) isSameSet: (mShip at: to)) ifFalse: [^ self]].

  mark := Array with: rel with: #oneSide.
  rel oneSide do: [:one | (mShip ocAt: one) add: mark].
  rel pairs keysDo: [:one | (mShip ocAt: one) add: mark].
```
♦ ⇨82⇦

8.5.2 Colouring

Algorithm 93. Composing paths.

The Loop to compose paths is straightforward:

```
  1 to: relLength - 1do: [:len | self relationshipsComposePlusOne: len].
```

The method used expands the paths of relationships with maximal path length by 1:

```
NetStructure>>relationshipsComposePlusOne: length
  | myPaths |

  myPaths := OrderedCollection new.
  self relationships keysAndValuesDo: [:path :rel |
    (path first ~= path last and: [path size = (length + 1)]) ifTrue: [
      myPaths add: path]].

  myPaths do: [:path |
    path first prePostNodes do: [:nd |
      self relationshipCompose:
        (Array with: nd with: path first) with: path].
    path last prePostNodes do: [:nd |
      self relationshipCompose:
        path with: (Array with: path last with: nd)]].
```

This method checks whether the two paths concatenate to a simple path and whether the resulting path has not been added yet. If so it composes the relationships. ♦ ⇨83⇦

Algorithm 94. Colour equivalence.

```
NetReduction>>selectBijectiveRelationshipsRelating: imageRel
   | connections |
   connections := OrderedCollection new.
   self structure relationships keysAndValuesDo: [:path :rel |
     ((imageRel is: path first equivalent: path last) not
         and: [rel isTo1
         and: [rel isFrom1
         and: [rel forward size = (self originsAllOf: path first) size
         and: [rel reverse size = (self originsAllOf: path last) size]
         ]]]) ifTrue: [
       imageRel relate: path first to: path last.
       connections add: rel]]].
^connections
```

This methods runs through all relationships. If the start and the end nodes of the path are not related yet and the relationship is a bijection between the sets of origins the two nodes then get related and the relationship is added to the collection of connecting relationships. ♦ ⇨83⇦

Algorithm 95. Colouring the nodes.

```
NetReduction>> colourNodesForEquivalence: imageRel
   | colSet origs root ima |

   imageRel  do: [:imaCla |
     origs := self originsAllOf: (ima := imaCla elements any).
     colSet := ColourSet new: origs size.
     root := self newRelationship source: colSet; dest: ima.
     colSet colours with: origs do: [:col :or |
       root origin: col image: or].
     self colourByImage at: ima put: root analyse].
```

This method only colours one node for each colour-set the remaining nodes get coloured along the connecting bijective relationships:

```
NetReduction>>colourNodesForConnectionCompositions: connections
   | atSource atDest |
   self colourByImage keys do: [:rootIma |
   <starting from rootIma traverse the graph of connections> do: [:conn |
     atSource := self colourByImage at: conn source ifAbsent: [nil].
     atDest := self colourByImage at: conn dest ifAbsent: [nil].
     atSource isNil
       ifTrue: [self colourByImage
         at: conn source
         put: (conn inverse ° atDest)]
       ifFalse: [self colourByImage
         at: conn dest
         put: (conn ° atSource)]]].    ♦ ⇨84⇦
```

8.6 Extensions, Variations and Applications

8.6.1 Choice

Algorithm 97. Computes all reductions using relational algebra.

The algorithm starts with a tree consisting only of the root and mutates it step by step into a tree representation of Φ :

(i) It arrives at a leaf n of the tree being built.
(ii) check all transitions around places that are related in $\rho(n)$, and try to merge them (analogous to `PlaceReduction>>mergeArcsInfo:` from Algorithm 79)
(iii) If a merger of two transitions is possible then each possible merger is represented as a relation and the union of these relations as $\{\psi\} \cup^{*} \Psi$ with a relation ψ and a set of relations Ψ. ψ and Ψ are normalised such that
(iv) $\{\psi\} \cup^{*} \Psi$ is minimal, i.e. as coarse as possible. This means that there are no redundancies with $\omega(n) \cup^{*} \rho(n)$
(v) ψ is as coarse as possible
(vi) Ψ is a set of disjoint relations
(vii) ψ is merged with $\omega(n)$ and Ψ is added to the set Δ.
(viii) The algorithm loops back to (ii). to search further transitions to merge around places related in (the modified) $\omega(n)$ until there no more mergers are possible.
(ix) Now, the set Δ contains the multiple choice parts of all mergers for this node. A representation
$\{\psi\} \cup^{*} \Psi = \bigcup^{*x}(\Delta)$
with a maximal relation ψ and a minimal disjoint set of relation Ψ normalised as above is computed. If ψ is non-trivial (that is not the identity) once more ψ is merged with $\rho(n)$, Δ is set to $\{\Psi\}$ and the algorithm loops back to (ii). to search for additional merges.
(x) If ψ is trivial Δ is transformed into the children of n (n was a leaf before).
(xi) the newly added children of n are processed recursively starting at (i).

The algorithm to be careful about the different relations. It has to search place reductions to merge in $\rho(n)$ and not in $\omega(n)$ because the reductions in $\omega(n)$ belong to the level above n which has fewer variations. But the check whether two transitions may be merged has to be done in $\omega(n) \cup^{*} \rho(n)$. Only here conflicts in the final reduction may be avoided. Further, one has to decide what to do with common consequences of different alternatives. Should they be identified in every step of the algorithm and transferred to parent nodes. Or alternatively, should they be rationalised only after the computation of the whole tree? ♦ ⇨88⇦

8.6.2 Component Detection

8.6.3 Using a Cliché Library

8.6.4 Flat Search

8.6.5 Reducing the Reduction

Algorithm 98. Merging colours and nodes.

In Algorithm 84 `Transition>>mergeMapsTo:info:` produces a collection of maps that `Transition>>tryToMerge:info:` applies to the current reduction. Now a filter is switched in between:

```
NetReduction>>maps: mapCollection filter: equivalence inColourWood: wood
  | conf lastEqui currEqui fallback res |
  lastEqui := equivalence.
  res := mapCollection select: [:map |
    currEqui := lastEqui copyCollection.
    conf := map keysAndValuesConform: [:from :to |
      fallback := wood currentChanges.
      currEqui relate: from to: to.
      (wood join: (self colourByImage at: from) source
        with: (self colourByImage at: to) source
        detect: [self isEquivalence: currEqui
                  disjointInColourWood: wood]
      ) ~= false].
    conf
      ifTrue: [lastEqui := currEqui]
      ifFalse: [wood undoChangesTo: fallback].
    conf].
^ res
```

For every map in the collection this method tries to find a consistent colouring. If none the changes already done for this map are undone before proceeding to the next map. There are two methods for undoing changes:

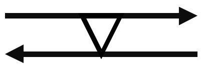
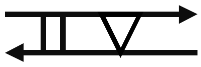
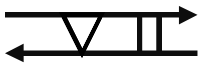
- For the equivalence a copy is used to keep the old state. In case of a fallback this copy is activated.
- In the colour wood changes are undone by removing arcs in the reverse add order.

```
ColourWood>>join: colour with: other detect: aBlock
  | emb othEmb fallback |
  (emb := self finalEmbeddingCompositionOfColour: colour) dest
      == (othEmb := self finalEmbeddingCompositionOfColour: other) dest
    ifTrue: [aBlock value ifTrue: [^ true]]
    ifFalse: [
      self
        colour1to1ConnectionsFromLeavesOf:
            (self trees nodeAt: emb dest)
        to: (self trees nodeAt: othEmb dest)
        do: [:conn |
          fallback := self currentChanges.
      self combineColours: conn.
      aBlock value ifTrue: [^ true].
      self undoChangesTo: fallback]].
^ false
```

This method tries the different paths between the leaves of the colours to be merged, until the caller is satisfied with a connection. ♦ ⇨92⇦

9 Index

The following table contains all symbols and terms used in this work except if they are elementary or used only locally.

symbol	**explanation**	➪
	$\begin{tikzcd} \underline{C} \arrow[r, shift left, "L"] \arrow[r, phantom, "\triangledown"] & \underline{D} \arrow[l, shift left, "R"] \end{tikzcd}$ An adjunction between category <u>C</u> and <u>D</u>. The triangle between the functor arrows points from the left adjoint L to the right adjoint R,	24
	$\begin{tikzcd} \underline{C} \arrow[r, shift left, "L"] \arrow[r, phantom, "\parallel\ \triangledown"] & \underline{D} \arrow[l, shift left, "R"] \end{tikzcd}$ A coreflection, the units Id→R L being isomorphic	24
	$\begin{tikzcd} \underline{C} \arrow[r, shift left, "L"] \arrow[r, phantom, "\triangledown\ \parallel"] & \underline{D} \arrow[l, shift left, "R"] \end{tikzcd}$ A reflection, the counits L R→Id being isomorphic	24
≤	Π' ≤ Π" for sets of relations means that for each relation in Π' there is a coarser relation in Π"	86
•	$^\bullet x$ is the preset and $x^\bullet$ the postset of a node x	
∪	set union. S∪{e} is abbreviated to S∪e if clear from the context	
→	f: X→Y designates a function, morphism, functor or natural transformation from X to Y	
∃!	unique existence quantifier: there exists one and only one.	102
∪*	π' ∪* π" is the minimal equivalence relation containing the tuples of both relations π' and π".	86
*	R^* is the reflexive and transitive (finite) closure of a relation R	
*MM2I	the functor *MM2 crumples a net saving only the initial marking and inventing transitions to allow morphisms.	45
<u>*ND</u>	the full subcategory of <u>*SYS</u> with neither dead transitions nor never marked places	46
<u>*NET</u>	any of the categories <u>PTNET</u>, <u>PPNET</u> or <u>FNET</u> usually used paired with the corresponding <u>*SYS</u> category.	44
<u>*SYS</u>	any of the categories <u>PTSYS</u>, <u>PPSYS</u> or <u>FSYS</u> usually used paired with the corresponding <u>*NET</u> category.	44
[>	m[σ>m' the occurrence of a step σ leading from marking m to m'. m[σ> means σ is enabled at m.	
[X, Y]	the set of morphisms <u>C</u>[X, Y] from object X to object Y in category <u>C</u>. If <u>C</u> is clear from the context it may be dropped.	
[x,y]	the interval of integers from x to y	
$[x]_R$	the equivalence class of x for the equivalence relation R	
|	restriction of a function or a relation	86

symbol	**explanation**	➩
| s |	the cardinality of a set or a multiset for example |x + 2y | = 3	
0	the zero element of a multiset or a vector space	
1S	1S: SETP→1S is the functor embedding sets into one-sets	25
1S	the category of one-sets having multisets as objects but only a subset of the linear functions as objects.	25
adjacency criterion	the reduction algorithm determines by an adjacency criterion which similar subnets to merge.	40
adjoint, adjunction	a relationship between two categories formed by two functors. One functor is called left adjoint and the other (in the reverse direction) the right adjoint.	
`allocate`	the transition in the spare-part ordering system representing that a part is allocated	56
`allocating`	the place in the spare-part ordering system representing the state of an order in which parts get allocated	56
`aPart`	the place in the spare-part ordering system representing allocated parts	56
`aPos`	the place in the spare-part ordering system representing allocated positions	56
B	B: 1S→SETP is the functor mapping a one-set to its base being the support	25
basic types	the basic subnets used in the reduction algorithm.	67
binary	a PTNET morphism fulfilling $f_\gamma X \subseteq \{0, 1\}$. Hence it corresponds to a partial function of the base sets.	27
BN	BN: 1S→SETP is the functor mapping a one-set to its base skeleton being the non zero multiples of the support	25
C*NET	one of the categories CPTNET, CPPNET, CFNET of coloured Petri nets.	36
CFNET	the category of coloured nets with foldings	36
clustering	a morphism or structuring principle that may collapse neighbourhoods in a single object	10
cocomplete	a category is cocomplete if the colimit of each diagram exists.	102
colimit	the limit of a diagram D is an object U and a transformation u: D→U which is natural and (co)universal	102
coloured net	a Petri net with nodes and arcs that are coloured by colour-sets. This allows a compact formulation of a large net.	36
commutative diagram	in category theory a diagram consisting of objects (nodes) and morphisms (arrows) with the composition of arrows corresponding the composition of morphisms.	102
commutative net diagram	a net diagram showing certain properties of relationships.	94
complete	a category is complete if the limit of each diagram exists.	102
coreflection	an adjunction with the units being all isomorphisms	24
counit	the natural transformation LR→ID with L a functor left adjoint to R	24
CPN	A special class of coloured nets from [Je92].	36

symbol	**explanation**	⇨
CPPNET	the category of coloured nets with place-preserving morphisms	36
CPTNET	the category of coloured place-transitions nets	36
^{d}	superscript for the destination of a coloured net C: $N^s \to N^d$	36
def	the elements of X of a function f: X→Y on which the partial function f is defined	
`defining`	the place in the spare-part ordering system representing the initial state of the life-cycle of an orders	56
Dictionary	A class from the Smalltalk class library implementing a function (from `key` to `value`) by a hash table.	
`done`	the place in the spare-part ordering system representing the final state of the life-cycle of an orders	56
DST	the functor mapping a coloured net to the destination	36
dst	the destination Y of a function f: X→Y	
e	e(r) is the number of edges currently in the arc map of the place reduction r	140
e	the number of edges of a graph or a net	
e_1	e_1(r) is the number of original edges merged until now into a place reduction r	140
env	the environment of a node that is the node itself plus its direct neighbours: env x = $^\bullet x \cup x \cup x^\bullet$	
F	the functor F: PPNET→PNET adds to each a place a transition	32
F	the flow relation of a net F ⊆ PxT ∪ TxP	
`finish`	the transition in the spare-part ordering system changing the state of an order from `packing` to `done`.	56
FNET	the category of nets with foldings as morphisms	33
folding	a PTNET morphism fulfilling f_β P⊆P' and f_β T⊆T'. This means mapping places to places and transitions to transitions.	10
FSYS	the category of systems with foldings as morphisms	44
full subcategory	A is a full subcategory of B if A is a subcategory of B and all objects X, Y of A have the same morphisms in both categories.	
f_β	is B f the retraction of a 1S morphism to a partial map between the base sets	25
f_γ	the coefficients of an f∈ 1S [X, Y] that is a function X→ℕ.	25
H*NET	the categories of hierarchical nets, HPTNET, HPPNET or HFNET	37
hierarchical net	A Petri net class which may express hierarchical structuring	
ic	instruction counter	63
Id	identity morphism or relation. If necessary the object or relation is indicated by a subscript.	
ID	Identity functor or functor ID: *NET→C*NET mapping an object X to Id_X.	36
IM	the functor IM: *SYS→*SYS maps restricts a system to the places o f the support of the initial marking.	44
im	the image of a function f: X→Y being the subset f X⊆Y	

symbol	**explanation**	⇨
invariant	a net invariant is either a place invariant or a transition invariant.	31
IP	the functor IP: $\underline{*SYS} \rightarrow \underline{*NET}$ maps a system to the net consisting of the places of the support of the initial marking.	44
I_S	the initial marking of a system S. The subscript S may be dropped.	
li-component	a net or subnet which cannot be reduced by local injections.	68
limit	the limit of a diagram D is an object U and a transformation u: U→D which is natural and universal	102
local injective	a morphism which is injective on the environment of each (single) transition.	34
log	the binary logarithm	
maxAuto	the maximal number of automorphisms of a connected single transition subnet	
maxDeg	maximal degree. The maximal number of (incoming plus outgoing) arcs of a node in a graph or a net	78
maximal ι reduction	a morphism from a net to a smaller net with certain universal properties	38
MM2	the functor MM2 keeps the places of net and adds transitions for each combination of pre and post.	33
MS	MS: $\underline{SETP} \rightarrow \underline{MS}$ the functor embedding sets in multisets	24
$\underline{MS}$	the category of multisets and linear functions.	24
$\mathbb{N}$	the natural numbers including zero	
$\mathbb{N}^+$	the naturals numbers excluding zero	
natural	a synonym for commutative	102
natural equivalence	a natural transformation with all η_X isomorphic.	
natural transformation	η: F→G is a natural transformation for two parallel functors F and G iff η_X: FX→GX is a morphism such that for each morphisms f holds $\eta_{\text{dst } f}\, F\, f = G\, f\, \eta_{\text{src } f}$	
ND	the functor ND: $\underline{*SYS} \rightarrow \underline{*ND}$ removes dead transitions and never marked places.	46
neighbourhood criterion	the reduction algorithm determines by an adjacency criterion which similar subnets to merge.	40
NET	the functor NET: $\underline{*SYS} \rightarrow \underline{*NET}$ maps a system to the underlying net.	44
N^P	the spare part system as a net coloured only with parts	60
N^{POSB}	the spare part system as a net coloured only with parts, orders, shipments and boxes	60
N^u	the spare part system as uncoloured net N^u	56
o	o(r) is the size of op(r).	140
O	in the order of magnitude of. g = O(f) means there are coefficients $\alpha, \beta \in \mathbb{N}$ with $g \leq \alpha\, f + \beta$.	
OCC	the functor OCC: $\underline{FSYS1} \rightarrow \underline{OCC}$ unfolds a system in an occurrence system.	51

symbol	**explanation**	➪
<u>OCC</u>	the category of safe occurrence systems is a full subcategory of <u>WOCC</u>.	48
op	op(r) is the set of initial place reductions that are currently merged into the place reduction r.	140
`oPart`	the place in the spare-part ordering system representing orderable parts	56
`oPos`	the place in the spare-part ordering system representing orderable positions	56
P	the functor P: <u>PPNET</u>→<u>PPNET</u> drops the transitions and keeps the places of a net	33
P, P_N	the set of places of a net N. The subscript N may be dropped.	33
`pack`	the transition in the spare-part ordering system representing that a part is packed	56
`pack`	the transition in the spare-part ordering system representing that a part is packed	56
`packing`	the place in the spare-part ordering system representing the state of an order in which parts get packed	56
part	one of the places `oPart`, `aPart`, `sPart` in the spare-part ordering system	56
PL	the functor PL: <u>PPNET</u>→<u>PPNET</u> forgetting transitions and keeping the Places	33
place invariant	a linear function from the place multisets of a net to the integers which does not change under transition occurrence.	31
place-preserving	a morphism is place-preserving if it maps places to places. <u>PPNET</u> is the category of nets and place-preserving morphims.	32
place-transition net	a Petri net that is an object of the category <u>PTNET</u> which allows clustering and folding	27
P_N	the set of places of a net N. The subscript N may be dropped.	
position	one of the places `oPos`, `aPos`, `sPos`, `pPos` in the spare-part ordering system	56
post	the post function of net assigning to every transition the multiset of post places	27
PP	the functor PP: <u>PTNET</u>→<u>PPNET</u> replaces each transition by three transitions and one place	32
pp	the combined pre and post function: pp = (pre, post)	32
<u>PPNET</u>	the category of nets and place-preserving morphisms.	32
`pPos`	the place in the spare-part ordering system representing packed positions	56
<u>PPSYS</u>	the category of systems with place-preserving morphisms	44
pre	the pre function of net assigning to every transition the multiset of pre places	27
<u>PROC</u>	the category of processes is a full subcategory of <u>OCC</u>.	48
PTNET	the functor PTNET: <u>1S</u>→<u>PTNET</u> maps a multiset to a net having the multiset base as places and not transitions	29
<u>PTNET</u>	the category of place-transition nets with morphisms allowing clustering and folding	27

symbol	**explanation**	➩
<u>PTSYS</u>	the category of place-transition systems	44
$\mathbb{Q}$	the rational numbers	
reachability graph	the reachability graph of a Petri system consists of the reachable markings (nodes) and enabled transitions (arcs).	
reduction	a morphism from a net to a smaller net with certain natural properties	38
reflection	an adjunction with the counits being all isomorphisms	24
^{s}	superscript for the source of a coloured net C: $N^s \to N^d$	36
semi positive	all coefficients being non negative	
<u>SET</u>	the category of sets and total functions	
<u>SETP</u>	the category of sets and partial functions	24
`ship`	the transition in the spare-part ordering system representing that a part is shipped	56
`shipping`	the place in the spare-part ordering system representing the state of an order in which parts get shipped	56
SM	the functor SM: <u>PPSYS</u>→<u>SM</u> unfolds a system into it step reachability graph (state machine).	46
<u>SM</u>	the category of state machines is a full subcategory of <u>PPSYS</u>.	46
small transition property	a net has the small transition property iff the cardinality of transition pre and postsets are limited by a constant Γ.	68
Smalltalk	an object-oriented language.	132
spare-part order system	an application serving as running example.	56
`sPart`	the place in the spare-part ordering system representing shipped parts	56
`sPos`	the place in the spare-part ordering system representing shipped positions	56
SRC	the functor mapping a coloured net to the source	36
src	the source X of a function f: X→Y	
`startA`	the transition in the spare-part ordering system changing the state of an order from `defining` to `allocating`.	56
`startP`	the transition in the spare-part ordering system changing the state of an order from `shipping` to `packing`.	56
`startS`	the transition in the spare-part ordering system changing the state of an order from `allocating` to `shipping`.	56
state machine	a Petri net system with a single token which is preserved by all transition occurrences.	46
step	a multiset of transitions of a Petri net.	
subcategory	<u>A</u> is a subcategory of <u>B</u> if the objects and morphisms of <u>A</u> are objects and morphisms respectively of <u>B</u>.	
supp	the support of a multiset being the set of elements with non zero cardinality	
SYS0	the functor SYS0: <u>*NET</u>→<u>*SYS</u> maps a net to the system with the same net and initial marking **0**.	44
telephone net	An example of a telephone designed with Petri nets	65

symbol	**explanation**	⇨
T_N	the set of transitions of a net N. The subscript N may be dropped.	
transition invariant	a multiset of transitions σ is a transition invariant if pre σ = post σ.	31
T-system	a Petri net with all arc cardinalities 1 and $\lvert {}^{\bullet}p \rvert = 1 = \lvert p^{\bullet} \rvert$ for all places, hence T is a transition invariant.	31
types	the basic subnets used in the reduction algorithm.	67
U	the underlying functor used for various categories	
<u>UFNET</u>	the category of nets with unitary foldings as morphisms	34
undef	f x = undef means the partial function f is not defined on the element x	24
unit	the natural transformation ID→RL with L a functor left adjoint to R	24
unitary	a <u>PTNET</u> morphism fulfilling $f_\gamma X \subseteq \{1\}$. Hence it corresponds to a total function of the base sets.	27
universal	an object through which every other natural object factorises uniquely	102
universal construction	either a colimit or a limit	102
VisualAge	a development environment which includes Smalltalk.	
WOCC	the functor WOCC: <u>FSYS</u>→<u>WOCC</u> unfolds a system in a weighted occurrence systems containing all processes.	49
<u>WOCC</u>	the category of weighted occurrence systems is a full subcategory of <u>FSYS</u>.	48
^{x}	Π' op^x Π'' for sets of relations means $\{\pi'$ op $\pi'' \mid \pi' \in \Pi'$ and $\pi'' \in \Pi''\}$ for any binary operator op on relations. Analogously for a set of relation sets.	86
x	A x B is the cartesian product of two sets A and B	
$\mathbb{Z}$	the integer numbers	
$\mathbb{Z}$-modules	A $\mathbb{Z}$-module over a set S is a function S→$\mathbb{Z}$. The category of $\mathbb{Z}$-module consists of finitely generated free Z-modules and linear functions.	
$\mathbb{Z}_\lambda$	the ring (or field) of the integers modulo λ	
Δ	the diagonal operator $\Delta x = (x,x)$	
Φ	the set of relations corresponding to the set of all reductions	147
Γ	A net has the small transition property iff the cardinality of transition pre and postsets are limited by a constant Γ.	68
Π	$\Pi(n)$ the set of relations at the subtree with root n	88
α	invocation count of PlaceReduction>>mergeArc:at:info:	139
$_\beta$	a <u>1S</u> morphism f_β = B f is the retraction to the base set	25
β	$\beta(r)$ is the accumulated number of invocations of PlaceReduction>>mergeArc:at:info: moving the $e_I(r)$ initial edges to the e(r) current edges of the place reduction r.	139

symbol	**explanation**	⇨
γ	f_γ is the coefficient function of a 1S morphism	25
γ	a slowly increasing inverse of the Ackermann function	78
ι reduction	a morphism from a net to a smaller net with certain natural properties	38
π_B	the projection within the hierarchical net of the spare part system along the dimension of boxes	60
π_P	the projection within the hierarchical net of the spare part system along the dimension of parts	60
ρ	$\rho(n)$ the relation at node n	88
ω	$\omega(n)$ the relation from node n upward to the root	88

Document info. Version 2.4, 03.12.00 15:22, 294701 characters, 59698 words, 157 pages.

Curriculum Vitae

name	Walter Keller
birthday	April 1, 1956
bürgerort	Herisau AR
contact	wkeller@tiscalinet.ch, http://home.tiscalinet.ch/wkeller Steinackerweg 33, 8304 Wallisellen
1975	Matura C am Mathematischen Naturwissenschaftlichen Gymnasium in Zürich
1982	Diplom in Mathematik der Universität Zürich Diploma thesis: Fast Algorithms in Linear Algebra
1982 - 1990	Software engineer at Burroughs (later Unisys), Zürich and Thalwil
1990 - today	Holenweg & Partner, Küsnacht. Mandates for the following clients:
1990 - 1992	Sulzer Rüti, Rüti
1992 - 1998	Mobiliar Versicherung, Bern
1998 - 2000	Zürich Versicherungen, Zürich
2000 - today	Zürcher Kantonalbank, Zürich