Relating Event-driven Process Chains  
to Boolean Petri Nets

Peter Langner[[1]](#footnote-1) Christoph Schneider[[2]](#footnote-2) Joachim Wehler[[3]](#footnote-3)

**Abstract**: One of the widespread methods for modelling business processes is the method of event-driven process chains (EPCs) (Ereignisgesteuerte Prozeßkette [EPK]). EPCs can be translated into Boolean nets, a class of coloured Petri nets with a single colour of type *Boole* and formulas from propositional logic as guards. The structure of the resulting Boolean net is a tree of bipolar synchronisation graphs (bp graphs). This property simplifies considerably the behavioural analysis of EPCs, because Genrich and Thiagarajan proved, that well-formedness of bp schemes can be tested by a reduction algorithm. If the Boolean net resulting from the translation of an EPC is well-formed it can be eventually translated into a free-choice net showing the same behaviour. Therefore the translation of EPCs into Boolean Petri nets fixes the semantic of EPCs, allows a formal analysis of the EPC-method and provides a well-founded base for taking further steps applying EPCs in the context of animation and simulation. In the domain of business process engineering only those EPCs, which have been certified as well-formed, can be recommended for further steps like simulation, activity based cost analysis or workflow.

1 Translation of EPCs into Boolean nets

1.1 Event-driven process chain (Example)

1.2 Event-driven process chain (Definition)

1.3 Syntax of EPCs (Remark)

1.4 Semantic of EPCs (Remark)

1.5 Boolean net (Definition)

1.6 Translation of EPCs into Boolean nets (Procedure)

1.7 Boolean net of an EPC (Definition)

1.8 Heuristic of the translation (Remark)

2 Boolean guards

2.1 Boolean transition of type xor, and, or (Definition)

2.2 Bindings of a Boolean transition (Remark)

2.3 Elementary logical alternative (Definition)

2.4 Branch/fork resolution (Definition)

2.5 Faithfulness concerning activation (Definition)

2.6 Boolean polynomials (Remark)

2.7 Logic and Boolean polynomials (Lemma)

2.8 Fusion of Boolean transitions (Definition)

2.9 Binding elements after fusion (Proposition)

2.10 Split and fusion of Boolean transitions (Proposition)

3 Boolean loop trees

3.1 Adjunction of pointed nets (Definition)

3.2 Loop tree (Definition)

3.3 Loop tree (Remark)

3.4 Siphons, traps and P-components (Proposition)

3.5 Well-formedness of a loop tree (Theorem)

3.6 Boolean loop tree (Definition)

3.7 Base marking of a Boolean loop tree (Corollary)

3.8 Cyclization and decyclization (Definition)

4 Analysis of Boolean nets and EPCs

4.1 Well-structuredness of Boolean loop trees and EPCs (Definition)

4.2 Or-well-structured elementary Boolean loops (Remark)

4.3 Well-formedness of Boolean loop trees and EPCs (Definition)

4.4 Well-formed resp. or-well-structured Boolean loop trees (Proposition)

4.5 Well-formedness with respect to place refinement (Proposition)

4.6 Or-well-structured and well-formed elementary Boolean loops (Remark)

4.7 Bp schemes (Remark)

4.8 Or-well-structured and well-formed Boolean loop trees (Algorithm)

4.9 Branch/fork resolution of or-well-structured Boolean loop trees (Remark)

4.10 Certification of EPCs as well-formed (Remark)

4.11 Well-formedness of the EPC „Ordering“ (Example)

4.12 EPCs and free-choice net systems (Remark)

5 Conclusion and relation to other work

Introduction

In the wake of standard software systems like SAP/R3 or Baan Triton different methods and tools for the modelling of business processes flourish on the market. Some of them use Petri nets, other do not. In Germany the method of *event-driven process chains* (EPCs) ([Sch1994]) is one of the most widespread methods used in commercial projects. In a continuously increasing variety of projects this method serves for different purposes: To model business processes, to document industrial reference models ([SAPa1996]) and also to design workflows ([SAPb1996]).

But in spite of their wide spreading and their acceptance by customers EPCs suffer from a serious drawback: Their lack of formal rigor, neither the syntax nor the semantic of an EPC is well defined. This fault will become manifest at least when there is real need to check EPCs for their consistency in order to control workflow systems.

Therefore we have set out to provide EPCs with an exact syntax and semantic, to define the concept of a *well-formed* EPC and to look out for an algorithm for the verification of well-formedness. We show in this paper, that all these goals can be reached by translating EPCs into Petri nets and applying Petri net theory.

Concerning their expressive power EPCs correspond to a rather simple class of Petri nets, which we call Boolean nets. A Boolean net is a coloured Petri net, particularly well-suited for modelling the control flow of a system: The tokens of a Boolean net carry a single colour with the two values *true* resp. *false*. They represent activation resp. explicit deactivation of places. Branching and alternatives of the control flow are modelled by using formulas of propositional logic as guards.

We consider Boolean nets an interesting net type due to the following reasons:

* Boolean nets strengthen EPCs by Petri net theory. The benefits are: Boolean nets provide a formal syntax and semantic for the EPC-method, which grew up apart from theoretical computer science in the domain of business process engineering.
* Boolean net systems, which result from the translation of EPCs, provide the computer scientist with examples of bipolar synchronisation schemes (bp schemes) from the field of applications. The advantage of bp schemes: Their well-behavedness can be verified by a reduction algorithms without any need to consider the case graph. In addition, the synthesis problem has been solved for well-formed bp schemes by Genrich-Thiagarajan.
* Many projects, which aim at the improvement of business processes, use EPCs as a language for process specification. But the formal correctness of this specification is a precondition to execute simulations and to derive reasonable decisions from an activity based cost analysis. After translation of EPCs into Boolean nets the certification of EPCs is possible.
* And finally a motivation for the mathematician: The propositional calculus for the guards of a Boolean net allows to study this net class with the help of polynomial algebras over the field of two elements.

The following Figure 1 illustrates in a symbolic way the connection between EPCs and Boolean nets and our proposal, how to obtain well-formed process models.



Figure 1 Quality assurance for EPCs

# Translation of EPCs into Boolean nets

EPCs model the flow of control using three different types of net elements:

* Events
* functions
* and logical connectors.

EPCs have been invented by Keller, Nüttgens and Scheer ([KNS1991]). Nüttgens characterizes the EPK-method as follows*: “The EPC-method is based on Petri net theory in the main ... it can be considered as a kind of condition-event nets enriched by logical connectors.“* ([Nüt1995])

Boolean nets are a simple class of coloured Petri nets, but sufficient to model the control flow of a system. Like other coloured nets they have an underlying p/t net, of which the Boolean structure is just an enrichment by some logical constructs, namely by

* two kinds of tokens modelling *true* and *false*
* formulas of propositional logic serving as guards of the transitions.

In this chapter we define Boolean nets and translate EPCs into Boolean nets.

## Event-driven process chain (Example)

Throughout the whole paper we consider a fictious model of the business process „Ordering“, which is represented as an event-driven process chain in Figure 2.



Figure 2 Event-driven process chain „Ordering“

Obviously the reader catches an intuitive understanding of the process. This is the great advantage of the EPC-method. It makes clear, why the method gained quick acceptance from consultants and customers in commercial projects and proves good in daily work.

On the other hand, a formal syntax for EPCs is lacking up to now. Therefore, we propose the following Definition 1.2, which we had to reconstruct from the original paper [KNS1991] as well as from examples in the literature.

## Event-driven process chain (Definition)

A connected directed graph EPC = ( N, A ) is called *event-driven process chain (EPC)* iff it satisfies the following properties:

* The set N of nodes is the union of three pairwise disjoint sets E ≠ ∅ (events), F ≠ ∅ (functions), K (connectors of type *xor*, *or*, *and).*
* Every element from the set A of arcs connects two nodes of different types.
* Only nodes from K branch. All nodes in the preset of a connector belong to the same type and all nodes in the postset of a connector belong to the same type. The preset type is different from the postset type.
* All nodes at the border of EPC belong to E, there exists at least one start event without input arcs and at least one final event without output arcs.

## Syntax of EPCs (Remark)

We do not restrict the number of input resp. output arcs of a logical connector, but we exclude any hybrid, merging two logical connectors of different types into a single one. We will show in Proposition 2.10 that any logical connector can be reduced to a subnet consisting of *binary* connectors, which have only two arcs with equal orientation and a third one with the opposite orientation.

## Semantic of EPCs (Remark)

1. Besides the lack of a formal syntax the semantic of an EPC is undefined, too. This shows up at the closing logical connectors. At a closing *and-*connector the process has to wait until all elements of the preset happen, but how to interpret closing *or-*connectors resp. *xor*-connectors?

* Is the closing *or-*connector in Figure 3 allowed to occur, when the first pre-event has happened, and to occur a second time for every following event? What does it mean in the model: the *first*?
* Presumably the closing *xor*-connector in Figure 3 shall be in deadlock, when both pre-events are activated at the same time. But what does it mean: at the *same* time?



Figure 3 Closing or- resp. xor-connector

EPCs do not model a global time or the concept of simultaneity, hence the transition rule for the closing *or*‑connector and for the closing *xor*‑connector is undefined.

2. Questions like the aforementioned have not been treated in the original sources about EPCs ([KNS1991]). In the meantime the semantic of a closing *or*- resp. *xor*-connector has been fixed in a pragmatic way by the implementation of EPCs in SIMPLE++[[4]](#footnote-4). This simulation tool, which forms part of release 3.2 of ARIS-Toolset[[5]](#footnote-5), uses a global time: The closing *xor*-connector is in deadlock if both events happen at the same time, while it occurs two times if they happen in sequence. Similarly the closing *or*-connector fires only once, if both events happen at the same time, whereas it occurs two times if they happen in sequence.

3. In the present paper we follow a different approach: We translate an EPC into a Petri net by a formal procedure and define the semantic of the EPC as the semantic of the resulting Petri net.

In [LSW1997] we introduced a class of coloured Petri nets - called *Boolean nets* - which are powerful enough to represent the functionality of EPCs. For Boolean nets we consider Boolean markings, which induce live and 1-safe markings on the underlying p/t net, if we forget about the colour of all Boolean tokens. Because we require the resulting p/t system to be live and 1-safe, any deadlock of a Boolean net system is due to the logic embodied in the guard functions and the token values, but it does not originate from the underlying p/t system. Concerning the notation for coloured nets cf. [Jen1992].

## Boolean net (Definition)

i) A *Boolean net* BN = (N, x, g) is a coloured Petri net consisting of a place/transition net N = (P, T, A) annotated in the language BOOLE of Boolean expressions. The annotation consists of

* an arc annotation



assigning a Boolean variable x(a) to every arc a ∈ A,

* and a transition annotation



assigning a Boolean expression gt, the guard formula, to every transition t ∈ T. The free variables of the guard formula are the variables annotating the input and output arcs of t. A transition of N together with its guard formula is called *Boolean transition* of BN. The net has a single tokencolour Boole = { true, false }.

ii) Depending on the type of the underlying place/transition net we define the subclasses of *Boolean free-choice nets*, *Boolean T-nets* resp. *Boolean P-nets*.

iii) A *Boolean marking* BM of a Boolean net BN is a mapping

BM = (BM0, BM1): P → ***N***2

inducing a live and 1-safe marking M of the underlying place/transition net N

M: P → ***N***, M(p) := BM0(p) + BM1(p).

iv) A *Boolean net system* BNS = (BN, BM) consists of a Boolean net BN provided with an initial Boolean marking BM.

## Translation of EPCs into Boolean nets (Procedure)

The translation of a given EPC into a Boolean net proceeds along the following four steps:

1. If the EPC contains a circuit we have to check in the first step, if the circuit models a loop and which *xor*-connector model the begin resp. the end of the loop. We call these connectors the *articulation* points of the loop, cf. Definition 3.2.

Loops with more than one begin resp. with more than one end are considered as error and have to be corrected by the modeler, who makes use of his process know-how. Similarly we consider it a mistake, to model loops with two articulation points, i.e. loops with begin different from their end. But the second kind of error can easily be repaired in a formal way, because every loop can be translated into a loop with exit test on top of the body. If necessary the body of the loop has to be duplicated and positioned before the articulation point a second time.

The example from Figure 2 contains a circuit, which models the complaint loop for those goods, which failed the quality test. This loop begins at the xor-connector K3 and ends at the xor-connector K2, hence it lacks a single articulation point. We translate the loop into a loop with a single articulation point after having recognised the following two facts:

* The body of the loop is formed by the sequence „E8, F5, E4, F3, E6, F4, K3“
* The sequence „F3, E6, F4, K3“ shall be executed before checking the loop condition for the first time.

The corrected EPC is given in Figure 4.



Figure 4 Process "Ordering" after loop correction

2. The second step translates the net elements of the resulting EPC according to the rules in the following Table 1.

| **EPC** | **Boolean net** |
| --- | --- |
| event | place |
| function | transition |
| logical connector of type *xor*, *or*, *and* | Boolean transition of the same type |
| *xor-*connector as articulation point | place |

Table 1 Translation rules

The guards for Boolean transitions of type *xor*, *or* resp. *and* will be introduced in the next chapter, cf. Definition 2.1.

3. During the third step one ensures that every articulation point is followed by transitions and adds some places to the resulting graph in order to satisfy the syntax of Petri nets.

4. In the final step one adds a distinguished place, called *start/end* or *basepoint* to the resulting Boolean net,and connectsevery place, which corresponds to a start event of the original EPC, with the basepoint. If the given EPC has more than one start event one has to model carefully the different possible combinations of the start events of the EPC with the help of Boolean transitions. Similarly one connects to the basepoint every desired combination of the final events of the EPC.

## Boolean net of an EPC (Definition)

We call the Boolean net, which results from the translation of an EPC according to Procedure 1.6, the *Boolean net of the EPC*.

## Heuristic of the translation (Remark)

1. We decided to translate the logical connectors of an EPC into Boolean transitions, in order to use the expressive power of the guards formulas to represent arbitrary logical formulas.

The occurrence rule of a Petri net requires *all* places in the preset of an activated transition to be marked. In particular, for the activation of a closing *xor*-connector, exactly one place of the preset has to be marked with an activation token, while all other places in the preset carry tokens representing deactivation. Therefore we introduce two different types of token: The token colour has two values *true* and *false*. Token with logical value *true* represent the activation of places, while token with logical value *false* represent the explicit *deactivation* of a place.

While EPCs lack any net elements marking the actual position of the control flow, the introduction of Boolean tokens resolves the indeterminacy in the semantic of the closing connectors. The closing *or*-connector and the closing *xor*-connector inherit their semantic from the occurrence rule of coloured Petri nets: First one has to wait for the information about the activation or deactivation of every place in the preset, secondly the guard formula decides on the base of the collected information if the control flow is allowed to pass the connector or not. In any case, for a logical connector to occur a second time *all* places of the preset have to be activated resp. deactivated a second time.

2. Considered in the light of our final Remark 4.12 the introduction of token with logical value *false* appears as a technical step in order to define the semantic of an EPC and to perform a net analysis in the context of Boolean nets. Using the results of Genrich and Thiagarajan every Boolean net resulting from a well-formed EPC can be simplified to a live and 1-safe free-choice net having neither annotations nor *false*-token.

3. Looking back from the ground of the Boolean net model for EPCs it seems to be a drawback of the EPC syntax to use the *xor-*connector in two different meanings, namely as logical connector, which opens resp. closes an *alternative*, and as articulation point of an inner *loop*, too.

In Figure 4 the *xor*-connector K3 is the articulation point of the complaint loop, while the *xor*-connectors K1 and K5 open resp. close the two different alternatives of ordering. A Boolean token entering K3 leaves the connector using only one of the two output arcs, while a Boolean token entering K1 splits into two tokens, which leave simultaneously using both output arcs. In the former case the control flow remains undivided, whereas in the latter it splits into two threads, which are synchronised at K5.

# Boolean guards

The guard formulas of Boolean nets, which arise from the translation of EPCs, use only on a subset of all formulas from propositional logic. They focus on the logical operators *xor*, *and* resp. *or*, an explicit negation is not part of the guard formulas. This expresses the fact, that an EPC does not model any spontaneous activation or deactivation of the control flow. To capture this property we introduce the concept of *faithfulness concerning activation*.

The well known fact, that the Boolean algebra of propositional logic has the algebraic structure of the field ***F***2 with two elements, allows to represent the guard formulas as polynomials over ***F***2 and to study these polynomials within the realm of commutative algebra.

## Boolean transition of type xor, and, or (Definition)

The logical type of a Boolean transition is determined by its guard formula: A Boolean transition with input variables xi, i =  1,..., n, and output variables yj, j =  1,..., m has *logical type* *xor* resp. *or* resp. *and* iff it has the guard formula

,

where 

## Bindings of a Boolean transition (Remark)

Depending on its logical type a Boolean transition t with input variables xi, i = 1,..., n, and output variables yj, j = 1,..., m, has the following bindings b ∈ Boolen+m, cf. Table 2:

|  |  |
| --- | --- |
| **Logical type** | **Bindings** |
| xor | * b = 0 := (0,...,0) * b = (bx1,...,bxn, by1,...,bym) with bxi = 1, byj = 1 for a unique i = 1,..., n and a unique j = 1,..., m |
| and | * b = 0 * b = 1 := (1,...,1) |
| or | * b = 0 * b = (bx1,...,bxn, by1,...,bym) with bxi = 1, byj = 1 for at least one i = 1,..., n and at least one j =1,..., m |

Table 2 Bindings of Boolean transitions

NB. We will use the algebraic value 1 as synonym for the logical value *true* resp. the algebraic value 0 for the logical value *false*.

## Elementary logical alternative (Definition)

1. We call a Boolean transition with a single input arc an *opening* transition and use the notation

* *branch-* (resp. *fork-* resp. *branch/fork*-) transition

for an opening transition of logical type *xor* (resp. *and* resp. *or*), cf. Figure 5.

Analogously we call a Boolean transition with a single output arc a *closing* transition and use the notation

* *merge-* resp. *join-* resp. *merge/join*-transition

for a closing transition of logical type *xor* resp. *and* resp. *or*.

2. We call the Boolean net of Figure 6 a *n-ary elementary logical alternative* of type *xor* resp. *and* resp. *or* iff the Boolean transitions (t1,t2) are of type (branch, merge) resp. (fork, join) resp. (branch/fork, merge/join).



Figure 5 Branch-transition



Figure 6 n-ary elementary logical alternative



Figure 7 Branch/fork resolution

Due to the fact, that the *or*-operator of propositional logic can be expressed as a combination of the *and*-operator and the *xor*-operator, every Boolean transition of type *or* can be eliminated. We call this substitution the branch/fork resolution.

## Branch/fork resolution (Definition)

According to the logical formula



every binary elementary *or*-alternative can be resolved into a series of branch- and fork-alternatives, its *branch/fork resolution*, according to Figure 7.

## Faithfulness concerning activation (Definition)

A Boolean net BN is called *faithful concerning activation* iff it has no spontaneous activations or deactivations, i.e. if every binding b of a Boolean transition t of BN satisfies:

If b = (bx, by) ∈ Boolen+m with respect to the input variables x = (x1..,xn) and the output variables y = (y1...,ym) of t then

bx = 0 <=> by = 0.

## Boolean polynomials (Remark)

1. One has the well known isomorphy between the Boolean algebra of two elements and the field ***F***2 with two elements

(Boole, *xor*, *and*) ≅ (***F***2, +, ·).

Moreover the algebra of Boolean functions



is isomorphic to the quotient



of the algebra of polynomials in k variables by the ideal generated by the relations xi2 - xi, i = 1,...,k. The ring B(k) is a reduced Artin ring, and every element of B(k) is idempotent, ([AM1969], Chapter 8). Using the above isomorphy we can translate every guard formula in k variables into a *guard polynomial* from B(k). The guard formula takes the logical value *true* iff the corresponding polynomial has value 1.

2. As is well known every symmetric polynomial in k variables can be expressed as polynomial in the k elementary symmetric polynomials. We denote by

σk,i ( x ) ∈ B(k), x = ( x1,...,xk ), i ∈ { 1,...,k },

the elementary symmetric polynomial in k variables of degree i and by

σk ( x ) := Σi=1,...,k σk,i ( x )

the sum of all elementary symmetric polynomials of positive degree in *x* = ( *x1,...,xk )*. E.g. for k = 2 there are two elementary symmetric polynomials

σ2,1 (x, y) = x + y, σ2,2 (x, y) = x y and σ2 (x, y) = σ2,1 (x, y) + σ2,2 (x, y) = x + y + x y.

The elementary symmetric polynomials satisfy the relation

σk ( x1,..., xk-1, xk ) = xk σk-1 ( x1,..., xk-1 ) + σk-1 ( x1,..., xk-1 ),

in particular due to char ***F***2 = 2

σk ( x1,..., xk-1, 1 ) = 0, σk ( x1,..., xk-1, 0 ) = σk-1 ( x1,..., xk-1).

## Logic and Boolean polynomials (Lemma)

1. Translating the logical operations op ∈ { *xor*, *or*, *and*} into polynomials from B(2) we have

* xor (x, y) = x + y = σ2,1( x, y )
* and (x, y) = x y = σ2,2( x, y )
* or (x, y) = x + y + x y = σ2( x, y ).

2. More general: Considering the logical operations as polynomials from B(n) gives

* op =.

The *xor*-operation is characterised by the recursion

* op( x, xn ) = ( 1 + xn ) op ( x ) + xn ( 1 + σn-1( x ) ), x = ( x1,...,xn-1 ).

**Proof**. ad 2. The case op = *and* is trivial, while the case op = *or* follows from the property

σn(y) = 0 < = > y = 0.

The recursion for the case op = *xor* follows also from

x = 0 < = > 1 + σn-1( x ) = 1.

We show by induction that the equation

op ( x1,..., xn ) = Σi=1,...,k σn,2i-1 ( x1,..., xn ) where n = 2k or n = 2k - 1

satisfies the recursion. The two initial cases n = 2 resp. n = 3 are easily checked.

Induction step n = 2k-1 to n+1 = 2k:

op ( x, x2k ) = ( 1 + x2k ) Σi=1,...,k σn,2i-1 ( x ) + x2k ( 1 + σn( x ) ) =

Σi=1,...,k σn,2i-1 ( x ) + x2k Σi=1,...,k σn,2i-1 ( x ) + x2k + x2k Σj=1,...,2k-1 σn,j ( x ) =

x2k + σn,1 ( x ) + Σi=2,...,k σn,2i-1 ( x ) + x2k Σi=1,...,k-1 σn,2i ( x ) =

σn+1,1 ( x, x2k ) + Σi=2,...,k σn+1,2i-1 ( x, x2k ) = Σi=1,...,k σn+1,2i-1 ( x, x2k ).

Induction step n = 2k to n+1 = 2k+1:

op ( x, x2k+1 ) = ( 1 + x2k+1 ) Σi=1,...,k σn,2i-1 ( x ) + x2k+1 ( 1 + σn( x ) ) =

Σi=1,...,k σn,2i-1 ( x ) + x2k+1 Σi=1,...,k σn,2i-1 ( x ) + x2k+1 + x2k+1 Σj=1,...,2k σn,j ( x ) =

x2k+1 + σn,1 ( x ) + Σi=2,...,k σn,2i-1 ( x ) + x2k+1 Σi=1,...,k σn,2i ( x ) =

σn+1,1 ( x, x2k+1 ) + Σi=2,...,k+1 σn+1,2i-1 ( x, x2k+1 ) = Σi=1,...,k+1 σn+1,2i-1 ( x, x2k+1 ), QED.

3. A binary Boolean transition with three adjacent arcs has the following guard polynomial

g(x, y ,z) ∈ ***F***2 [x, y, z]

with respect to the variables *x*, *y* annotating the two equal directed arcs and the variable *z* annotating the third arc:

* Type *branch* resp. *merge*:

g(x, y, z) = 1 + σ3( x, y, z) + σ2,1(x, y) z = 1 + σ2(x, y) + z + σ2,2(x, y) z

* Type *fork* resp. *join*:

g(x, y, z) = 1 + σ3( x, y, z) + σ2,2(x, y) z = 1 + σ2(x, y) + z + σ2,1(x, y) z

* Type *branch/fork* resp. *merge/join*:

g(x, y, z) = 1 + σ3( x, y, z) + σ2(x, y) z = 1 + σ2(x, y) + z.

## Fusion of Boolean transitions (Definition)

Let BN = (N, x, g) be a Boolean net and t1, t2 two Boolean transitions of BN sharing a unique place p, i.e.

\*t1 ∩ t2\* = ∅, t1\* ∩ \*t2 = {p}, \*p = { t1 }, p\* = { t2 }.

Let

(x; z) = (x1,...,xn, z) resp. (y; z) = (y1,...,ym, z)

denote the variables annotating the arcs adjacent to t1 resp. t2, where z annotates the two distinguished arcs adjacent to the shared place p. Let

g1 (x; z) ∈***F***2[x; z] resp. g2 ∈ ***F***2[y; z]

denote the guard polynomials of t1 resp. t2. Then we define the guard polynomial gt of the Boolean transition t resulting from the *fusion* of t1 and t2, cf. Figure 8, as

gt (x, y) := σ2 ( g12 (x, y; 0), g12 (x, y; 1) ) ∈ ***F***2[x, y],

g12(x, y; z) := g1 (x; z) g2 (y; z).



Figure 8 Fusion of Boolean transitions

## Binding elements after fusion (Proposition)

The Boolean transition t resulting from the fusion of two Boolean transitions t1 and t2 according to Definition 2.8 has as bindings all the compatible bindings of t1 and t2, i.e. (b1, b2) is a binding of t iff there exists a common binding bp of the variables annotating the two arcs adjacent to the shared place p such that (b1,bp) is a binding of t1 and (b2,bp) is a binding of t2.

**Proof.** The Boolean transition t has by definition the guard polynomial

gt (x, y) = σ2 ( g12 (x, y; 0), g12 (x, y; 1) ).

Because σ2 has a single zero at the origin we get

gt (x, y) = 1 <=> g12 (x, y; 0) = 1 or g12 (x, y; 1) = 1.

By definition of g12 we have

g12 (x, y; 0) = 1 <=> g1 (x; 0) = 1 and g2 (y; 0) = 1

and analogously for g12 (x, y; 1), QED.

## Split and fusion of Boolean transitions (Proposition)

Every Boolean transition t with n input and m output arcs of type op ∈ { *xor*, *and*, *or* } is equivalent to the net from Nt of Figure 9 with only binary transitions of the same logical type.



Figure 9 Split and fusion of Boolean transition

**Proof**. The statement is invariant under reversal of arc-orientations. By induction it can be reduced to two special cases:

1. Fusion of a closing transition with a closing binary transition according to Figure 10
2. Fusion of a closing and an opening transition by a shared place according to Figure 8.



Figure 10 Fusion with a binary transition

ad 1. We set x = ( x1,..., xn-1 ). The guard polynomial of the Boolean transition t is given by

* gt( x, xn, y ) = 1 + σn+1 ( x, xn, y ) + op ( x, xn ) y.

The guard polynomials of the two split transitions are

* g1( x, z ) = 1 + σn ( x, z ) + op ( x ) z, g2( z, xn, y ) = 1 + σ3 ( z, xn, y ) + op ( z, xn ) y

In two variables we have

* op ( 1, u ) = , op ( 0, u ) = 

Finally we compute the guard gfus, which results from fusion of the split transitions according to Definition 2.8, as

* gfus( x, xn, y ) = σ2 ( g1( x, 0 ) g2( 0, xn, y ), g1( x, 1 ) g2( 1, xn, y ) ) =

σ2 ( [ 1 + σn-1 ( x ) ] [ 1 + σ2 ( xn, y ) + op ( 0, xn ) y ], op ( x ) op ( 1, xn ) y ).

In order to verify the equation gt= gfus we evaluate both sides separately for the case y = 0 resp. y = 1. For y = 0 we get independently of the logical type

* gfus( x, xn, 0 ) = σ2 ( [ 1 + σn-1 ( x ) ] [ 1 + xn ], 0 ) = 1 + σn-1 ( x ) + σn-1 ( x ) xn + xn =

1 + σn ( x, xn ) = 1 + σn+1 ( x, xn, 0 ) = gt( x, xn, 0 ).

For y = 1 we get

* gt( x, xn, 1 ) = op ( x, xn )

i) In the case op = *or* we have

op ( 0, xn ) = xn, op ( x ) = σn-1 ( x ), op ( 1, xn ) = 1.

We get for y = 1

gfus( x, xn, 1 ) = σ2 ( [ 1 + σn-1 ( x ) ] xn, σn-1 ( x ) ) =

xn + xn σn-1 ( x ) + σn-1 ( x ) + σn-1 ( x ) xn + σn-1 ( x ) xn = σn ( x, xn ) = op ( x, xn ).

ii) In the case op = x*or* we have

op ( 0, xn ) = xn, op ( x ) = σn-1,1 ( x ), op ( 1, xn ) = 1 + xn.

We get for y = 1

gfus( x, xn, 1 ) = σ2 ( [ 1 + σn-1 ( x ) ] xn, op ( x ) [ 1 + xn ] ) =

[ 1 + σn-1 ( x ) ] xn + op ( x ) [ 1 + xn ] = op ( x, xn ),

using xn ( 1 + xn ) = 0 and Proposition 0, part 2.

iii) In the case op = *and* we have

op ( 0, xn ) = 0, op ( x ) = σn-1,n-1 ( x ), op ( 1, xn ) = xn.

We get for y = 1

gfus( x, xn, 1 ) = σ2 ( 0, σn-1,n-1 ( x ) xn ) = σn-1,n-1 ( x ) xn = σn,n ( x, xn ) = op ( x, xn ).

ad 2. This case follows from Proposition 0, QED.

# Boolean loop trees

The analysis of a Boolean net can be divided into different parts. In the present chapter we analyse the structure of the underlying p/t net. Due to Definition 1.5 the well-formedness of the underlying p/t net is equivalent to the existence of a Boolean marking on a Boolean net. Those p/t nets, which result from the translation of well-structured EPCs, have a particular net structure: They form a tree of loops. We define a *loop tree* (Schleifenbaum) as a net, which results from the successive adjunction of loops in a prescribed manner, and prove that loop trees are always well-formed.

Due to our translation of EPCs into Boolean nets from Procedure 1.6 the resulting net has a distinguished place, which we call its basepoint.

## Adjunction of pointed nets (Definition)

i) A *pointed net* (N, p) is a net N with a distinguished place p, which is called its *basepoint*.

ii) Let Ni, i = 0,1, be two disjoint place/transition nets with distinguished places l1 ∈ N0, p1 ∈ N1. We denote by



the fusion of N0 and N1 at the places l1 and p1. In the case of pointed nets

(Ni, pi), i = 0,1, and l1 ≠ p0

we call the pointed net (N, p) with basepoint p = p0 the *adjunction* of (N1, p1) to (N0, p0) at the place l1 and write

.

## Loop tree (Definition)

i) A pointed T-net EL = (N, p) is called *elementary loop* iff N is strongly connected and N \ p is acyclic.

ii) A pointed net LT = (N, p0) is called a *loop tree* (Schleifenbaum) iff there exist

* elementary loops ELi = (Ni, pi), i = 0,...,n,
* and pairwise disjoint places li ≠ p0 of N with \*(li\*) = { li }, i = 1,...,n,

such that LT = LTn according to the inductive adjunction LT0 := EL0 and



iii) For a loop tree we call the subnets ELi, i = 0,...,n, the *loop components* of LT, the distinguished component EL0 is called the *root component* and the fusion places li, i = 1,..,n, are called *articulation* *points*.

## Loop tree (Remark)

1. Every loop tree is a tree: The loop components are the nodes, the loop component ELi is a direct successor of ELj iff the articulation point lj, adjoining ELj, belongs to ELi.

2. Due to the condition about the neighbourhood of its articulation points every loop tree is a free-choice net.

3. In the case of an elementary loop (N, p) the restriction of the precedence relation defines a partial order „≤“ on the acyclic net N \ p. Moreover, there exists a unique minimum

* tmin = min (N \ p)= p\*

and a unique maximum

* tmax = max (N \ p)= \*p.

For two nodes x, y ∈ N \ p with x ≤ y, we define the *distance*

d(x, y ) := min { length γ: γ a directed path in N \ p from x to y } and d( y, x ) := d( x, y ).

## Siphons, traps and P-components (Proposition)

Consider a loop tree LT with place set P and basepoint p0.

i) For a subset R ⊂ P of places we have the equivalence:

* R is a minimal siphon
* R is a minimal trap
* R is the set of places of a P-component.

In all these cases N(R,\*R), the subnet of LT generated by R and \*R, is covered by circuits.

1. Every siphon in LT contains the basepoint p0.

iii) LT is covered by P-components.

**Proof.** We use the characterisation of siphons and traps of a net N = ( P, T, A ) from [Lau1987]. Lautenbach introduces the concept of D-systems (resp. T-systems): An element Γ of the free monoid generated by the circuits of N

, γ circuit, n(γ) ∈ ***N*** and n(γ) = 0 for all but finite γ

is called a *circuit system* of N. The support of Γ is defined as the set of places, which are covered by at least one circuit γ of Γ, i.e.

supp( Γ ) := { p ∈ P: Γ(p) ≠ 0 }.

The circuit system Γ is called a *D-system* (resp. *T-system*) iff for each place p ∈ P there exists a number Γ(p) ∈ ***N*** such that every output (resp. input) arc a ∈ A of p is covered by Γ with the same multiplicity Γ(p). A D-system Γ is called *minimal*, iff

* the numbers (Γ(p))p∈P are pairwise prime
* there does not exist a D-system Γ’ with support supp(Γ’) ⊂ supp(Γ), but supp(Γ’) ≠ supp(Γ).

For a strongly connected net N Lautenbach proves ([Lau1987], Theorem 2.9 and Corollary 2.11): Assigning to every D-system Γ its support

Γ supp ( Γ )

defines a bijective map from the set of minimal D-systems to the set of minimal siphons of N. In order to define the inverse map Lautenbach constructs for a given minimal siphon R a circuit system Γ covering the subnet N(R,\*R). The construction proceeds via backtracing along the backward arcs of R. Using the Farkas theorem he proves that Γ is already a D-system. An analogous relationship holds between minimal T-systems and minimal traps.

ad i) Now let R be a minimal siphon in a loop tree LT and let Γ be the corresponding D-system. Then Γ satisfies the condition for a T-system, too: For unbranched places of LT there is nothing to show. Every branched place p is an articulation point adjoining an elementary loop EL. We denote by ain,1, ain,2 (resp. aout,1, aout,2) the two input (resp. two output arcs) of p and assume ain,2, aout,2 in EL. We have

Γ(ain,1) + Γ(ain,2) = Γ(aout,1) + Γ(aout,2), Γ(aout,1) = Γ(aout,2), because Γ is a D-system,

and

Γ(ain,2) = Γ(aout,2),

because every circuit of EL passes through the articulation point. Therefore

Γ(ain,1) = Γ(ain,2),

and Γ satisfies the condition on T-systems at the articulation point p, too. By the result of Lautenbach R is a minimal trap. Reversing the argument shows that every minimal trap of LT is a minimal siphon.

Even in a general net the places of a P-component form a minimal siphon. For the reverse direction consider a minimal siphon R of LT. As already proven R is a minimal trap, too. Therefore \*R = R\* and the subnet N(R,\*R) is a P-net containing \*R ∪ R\*. Due to the minimality of the corresponding D-system the subnet N(R,\*R) is connected. It is even strongly connected, because it is covered by circuits. Hence N(R,\*R) is an P-component.

ad ii) By the result of Lautenbach for a minimal siphon R the D-system N(R,\*R) is covered by circuits. Due to the definition of an elementary loop every circuit contained in a loop component EL of a loop tree LT passes through the basepoint p of EL. If p equals the articulation point adjoining EL we get from the inclusion

\*p ⊂ N(R,\*R)

a possibly second circuit covering the input arc of p not contained in EL. Iterating the argument we finally get a circuit from N(R,\*R) passing through the basepoint of LT.

ad iii) A given node x ≠ p0 is contained in a well determined loop component EL = ( N, p ) with x ≠ p. Due to the strong connectedness of EL and because N \ p is cycle free, there exists a unique circuit in EL covering x. If p ≠ p0 we iterate the argument, now starting from the place p and arriving at a second circuit covering p. After a finite number of steps we arrive at a circuit system Γ covering x. We extend Γ to a minimal D-system by adding successively finitely many circuits at the articulation points. By the result of Lautenbach and by part i) the support of Γ is an P-component, QED.

## Well-formedness of a loop tree (Theorem)

Every loop tree is well-formed, i.e. marking the net with a single token at the basepoint defines a live and 1-safe marking.

**Proof.** Denote by LT the given loop tree and by M0 the distinguished marking.

* By Commoner’s theorem a free-choice system is live iff every minimal siphon contains an initially marked trap ([DE1995], Chap. 4.3.)

This property is satisfied by (LT, M0) due to Proposition 3.4, part ii).

* A live free-choice system is 1-safe iff it is covered by P-components, which carry at most one token ([BD1990], Corollary 5.6):

This property is satisfied by (LT, M0) due to Proposition 3.4, part iii), QED.

## Boolean loop tree (Definition)

A Boolean net BN = ( N, x, g ) is called *Boolean loop tree* iff the underlying net N is a loop tree. Marking the basepoint of BN with a single token of value *true* defines the *basemarking* of BN.

## Base marking of a Boolean loop tree (Corollary)

The basemarking of a Boolean loop tree is a Boolean marking.

## Cyclization and decyclization (Definition)

i) Denote by EL = ( ( P, T, A ), p0 )  an elementary loop with basepoint p0. Choose two new places pi and pf not contained in P. The net N, which results from splitting p0 into an initial place pi and a terminal place pt, is called the *decyclization* of EL, i.e.

N := decyl (EL) = ( PN, TN, AN ) with

* places PN := ( P \ p0 ) ∪ { pi, pt },
* transitions TN := T
* and arcs AN := (A | N \ p) ∪ { (pi, p\*), (pt, \*p) }.

ii) Denote by N an acyclic T-net with two uniquely determined places

pi = min (N) and pt = max (N),

where *min* resp. *max* refer to the partial order defined by the precedence relation. The elementary loop EL with basepoint p0, which results from the fusion of pi and pt to a new place p0, is called the *cyclization* of N. We write EL = cyl (N).

iii) For elementary Boolean loops we define the concepts of cyclization and decyclization by the analogous concepts referring to the underlying p/t nets.

# Analysis of Boolean nets and EPCs

After the translation of EPCs into Boolean nets according to the procedure from chapter 1 and due to the results about these nets from chapter 2 and 3 we are now ready to analyse EPCs by means of Petri net theory.

We distinguish between a structural net analysis and a behavioural analysis. EPCs, which pass the first one, are qualified as *well-structured*: They are structured as loop tree and have either none or paired *or*‑transitions. Those, which in addition pass the behavioural analysis, are called *well-formed*: They are also free of deadlocks and have only live transitions. We present a reduction algorithm for both types of analysis, which extends the Genrich-Thiagarajan reduction for well-formed bp schemes.

Concerning the structure of EPCs we define:

## Well-structuredness of Boolean loop trees and EPCs (Definition)

i) The set of elementary Boolean loops, which are *or-well-structured*, is the smallest set ELows with the following properties:

* ELows contains every elementary Boolean loop without transitions of type *or.*
* ELows contains every elementary *or*-alternative (cf. Definition 2.3).
* For B1, B2 ∈ ELows also the refinement of a place of B1, which is different from the base point, by decyl (B2) belongs to ELows.

ii) A Boolean loop tree is *or-well-structured* iff every loop component belongs to ELows.

iii) An EPC is *well-structured* iff its Boolean net is an *or*-well-structured Boolean loop tree.

## Or-well-structured elementary Boolean loops (Remark)

1) Every *or*-transition of a logical alternative EL belongs to a pair ( tbf, tmj ) with a branch/fork transition tbf and a merge/join transition tmj. Setting pi := \*tbf and pt := tmj\* this pair has the following properties:

i) Both transitions tbf and tmj belong to the same loop component.

ii) If we denote by Γ( pi, pt ) the set of directed simple paths within EL from pi to p or from p to pt, then every γ ∈ Γ( pi, pt ) covers both places pi and pt.

iii) outdeg ( tbf ) = indeg ( tmj ) =: k.

iv) If we denote by N( tbf, tmj ) the subnet of EL, which is generated by all directed simple paths within EL from pi to pt, then

N( tbf, tmj ) \ { tbf, tmj, pi, pt }

splits into k different connectedness components.

2) The pair ( tbf, tmj ) and its properties i) - iv) do not change, neither when a place of EL, which is different from the base point, is refined by the decyclization of an element of ELows, nor when decyl (EL) itself is substituted as place refinement into another element of ELows. Hence the above remark 1) holds also for every elementary Boolean loop, which is *or*-well-structured.

3) In Definition 4.1 part iii) we required a separate condition about the *or*-connectors in order to qualify a given EPC as well-structured. Solely well-formedness of the corresponding Boolean loop tree would be too weak, to rule out some type of EPCs we consider to be ill-structured: E.g. a Boolean loop tree having only Boolean transitions of type *branch/fork* and *merge/join* is well-formed according to Theorem 3.5, nevertheless it can be ill-structured in our opinion: A lot of EPCs from the literature and from commercial projects in the field of business process engineering demonstrate that the unrestricted use of closing *or*-connectors does not model any real situation. Rather it reveals the failure of the modeler to synchronise in a correct way all alternatives he has created. A closing *or*-transition never generates any deadlock but often it has only been chosen to remedy a situation, which got out of control.

Concerning the behaviour of EPCs we define:

## Well-formedness of Boolean loop trees and EPCs (Definition)

i) A Boolean loop tree BLT, faithful concerning activation, is *well-formed* iff the Boolean net system (BLT, BM) is live concerning the basemarking BM.

ii) An EPC is *well-formed* iff it is well-structured and its Boolean loop tree is well-formed .

## Well-formed resp. or-well-structured Boolean loop trees (Proposition)

Denote by BLT a Boolean loop tree, which is faithful concerning activation.

1) BLT is well-formed iff the Boolean system (BLT, BM) is reversible and has no dead transitions, i.e. iff it satisfies the following two conditions:

* BM is a homespace (Reversibility)
* For every Boolean transition t of BLT there exists a reachable marking

BMpre ∈ [BM>

activating a binding b ≠ 0 of t (Non deadness).

2) BLT is well-formed (resp. *or*-well-structured and well-formed) iff every of its loop components is well-formed (resp. *or*-well-structured and well-formed).

**Proof.** ad 1. i) Assume BLT being well-formed and denote by p the basepoint of BLT. If t = \*p denotes the precondition of the basepoint there exists a reachable marking BMpre activating a binding element b ≠ 0 of t. The occurrence of b generates the successor marking BMpost with BMpost(p) ≠ 0.

But even BMpost = BM: Every place of the underlying loop tree LT is contained in a suitable P-component marked with at most a single token under the 1-safe marking

M := BM0 + BM1, BM = ( BM0, BM1 ),

cf. the proof of Theorem 3.5. But every P-component of LT contains the basepoint p by Proposition 3.4., hence Supp (BMpost) = { p }. Because BLT is faithful concerning activation we get BMpost = BM.

The second condition on the liveness of Boolean transitions obviously weakens the supposed liveness of (BLT, BM).

ii) Let t be a Boolean transition and BMpre a reachable marking of BLT. By assumption

BM ∈ [BMpre>,

and there exists a successor marking

BMpost ∈ [BM>

activating a binding element b ≠ 0 of t. Hence

BMpost ∈ [BMpre>

activates b, QED.

ad 2) The proof of the statement about well-formedness uses the result of part 1 and proceeds straightforward, but in a rather lengthy way. The additional statement about *or*-well-structuredness is just a paraphrase of Definition 4.1.

## Well-formedness with respect to place refinement (Proposition)

Assume BN to be an elementary Boolean net with an acyclic subnet N, which has two unique places pi = min (N) and pf = max (N) different from the basepoint of BN. Denote by BLT the Boolean loop tree, which results from BN by fusing both places pi and pf.

i) Then BLT has two loop components, the root component EL0 and a second component EL1 = cycl(N).

ii) We have the equivalence:

* BN is well-formed
* BLT is well-formed
* EL0 and cycl(N) are well-formed.

**Proof**. Part i) is obvious.

ad ii) The Boolean markings of BLT and BN correspond bijectively, because no Boolean marking of BN marks both places pi and pf. Under this correspondence also the markings, which are reachable from the basemarking of both systems (BLT, BMBLT) and (BN, BMBN), correspond bijectively. Therefore BLT is well-formed iff BN is well-formed, which proves the first equivalence,. The last equivalence follows from Proposition 4.4, part 2), QED.

By Proposition 4.4, part 2, we have reduced the question, if a given Boolean loop tree is *or*‑well-structured and well-formed, to the analogous question about an elementary Boolean loop. The following remark follows easily from Definition 4.1.

## Or-well-structured and well-formed elementary Boolean loops (Remark)

The class of *or*-well-structured and well-formed elementary Boolean loops is the smallest set ELowf with the following properties:

* ELowf contains every elementary Boolean loop without transitions of type *or.*
* ELowf contains every elementary *or*-alternative (cf. Definition 2.3).
* For B1, B2 ∈ ELowf also the place refinement of B1 by B2 at a place different from the base point of B1 belongs to ELowf.

The question about well-formedness of elementary Boolean loops, which have only transitions of logical type *and* resp. *xor*, has already been answered in a paper by Genrich and Thiagarajan ([GT1984]). They introduced a class of net systems, called *bipolar synchronisation schemes* (bp schemes), which turn out to be special Boolean net systems.

## Bp schemes (Remark)

i) A Boolean net system BNS = ( BN, BM ) is called *bipolar synchronisation scheme* (bp scheme) iff BN is a Boolean T-net and all transitions have logical type *xor* resp. *and.*

ii) For a bp scheme BP the following facts hold:

* BP is live iff BP is deadlockfree ([GT1984], Theorem 2.12)
* The synthesis problem for live bp schemes has been solved: BP is live iff it can be constructed from an elementary bp scheme, cf. Figure 11, by a kit of eight synthesis rules ([GT1984], Theorem 6.19)
* There exists a terminating reduction algorithm ([GT1984], Chapter 6.5) using six reduction rules with the property: BP is live iff it can be reduced to an elementary bp scheme from Figure 11.



Figure 11 Elementary bp schemes

## Or-well-structured and well-formed Boolean loop trees (Algorithm)

Input: Boolean loop tree BLT having only Boolean transitions of logical type *xor*, *or* resp. *and*.

Output: At successful termination „*or*-well-structured and well-formed“, at termination with error „not *or*-well-structured or not well-formed“.

Begin

Traverse the Boolean loop tree BLT post-order. For the current loop component ( EL, p ) of BLT do:

|  |  |
| --- | --- |
| 1 | * Denote by Tbf the set of branch/fork-transitions and by Tmj the set of merge/join-transitions of EL. |
| 2 | * Find an element tbf from Tbf with no successor transition from Tbf.   If Tbf = ∅:   * Check that Tmj = ∅, otherwise stop with error. * Check that the bp scheme EL is well-formed, otherwise stop with error. * Fold EL to its basepoint. * Exit. |
| 3 | * For tbf determine the set of nearest successor transitions from Tmj. * Check that there is exactly one such transition tmj, if not stop with error. |
| 4 | * For tmj determine the set of nearest ancestor transitions from Tbf ∪ Tmj. * Check that there is exactly one such transition, namely tbf, if not stop with error. |
| 5 | * Set pi := \*tbf and pt := tmj\* and denote by Γ( pi, pt ) the set of directed simple paths within EL from pi to p or from p to pt. * Check that every path γ ∈ Γ( pi, pt ) covers both places pi and pt, if not stop with error. |
| 6 | * Check that the outdegree of tbf equals the indegree tmj, if not stop with error. |

|  |  |
| --- | --- |
| 7 | * Denote by N( pi, pt ) the subnet of EL, which is generated by all directed simple paths within EL from pi to pt . * Check that the net   N( pi, pt ) \ { tbf, tmj, pi, pt }  splits into k := outdeg (tbf) different connectedness components Nj, j = 1,...,k, if not stop with error.   * Check that every bp scheme cycl(Nj), j = 1,...,k, is well-formed, if not stop with error. |
| 8 | * Replace the net N( pi, pt ) by a single place, which results as the fusion of pi and pt. * Denote by EL the resulting elementary loop. |
| 9 | * Repeat step 1. |

End.

**Correctness proof.**

* Step 7: It is checked, that every arc leaving tbf corresponds to a unique arc entering tmj and vice versa. The different threads do no interfere and the cyclization of every thread is a well-formed bp scheme.
* Step 2 and Step 7: Due to the Genrich-Thiagarajan reduction (cf. Remark 4.7) well-formedness of bp schemes can be checked without expanding the case graph.

The algorithm terminates: If it does not stop with error then it reduces the Boolean loop tree to its base point. For a given loop component each execution of the inner loop reduces either a pair ( tbf, tmj ) and its *or*-alternative or the whole component to a single place. Note that a non-empty set Tbf contains always a transition tbf without successor transitions from Tbf, because the elementary loop EL \ p is acyclic.

i) Assume BLT to be *or*-well-structured and well-formed. In order to prove, that the algorithm outputs „*or*-well-structured and well-formed“ we use Remark 4.2.

We show at first: If tbf has no successor transition of type branch/fork within EL, then tmj is the unique successor transition of tbf of type merge/join in EL with minimal distance.

For the proof we set tbf,1 := tbf, and assume two merge/join transitions tmj,1 := tmj and tmj,2 in EL with the same minimal distance from tbf,1. Denote by tbf,2 the branch/fork transition corresponding to tmj,2, cf. Figure 12. We consider the partial order on EL \ p and the corresponding distance introduced in Remark 3.3.

* We have tmj,2 ≤ tmj,1: Assume d(tbf,1, tmj,2) = length γ2 for a path γ2 from tbf,1 to tmj,2. Extend γ2 by a path γ3 from tmj,2 to p. Due to Remark 4.2 the composition γ3 o γ2 covers tmj,1. The first segment γ2 does not cover tmj,1, because otherwise d(tbf,1, tmj,1) < d(tbf,1, tmj,2). Hence the second segment γ3 covers tmj,1 and we get tmj,2 ≤ tmj,1.
* We have tbf,2 ≤ tbf,1: Choose a path γ4 from p to tbf,1 and extend it by composition with γ2 to tmj,2. Due to Remark 4.2 the composition γ2 o γ4 covers tbf,2. Therefore either tbf,1 ≤ tbf,2 or tbf,2 ≤ tbf,1. The first case is excluded because tbf,1 has no successor transition in Tbf.
* We have tmj,1 ≤ tmj,2: Assume d(tbf,1, tmj,1) = length γ1 for a path γ1 from tbf,1 to tmj,1. Because tbf,2 ≤ tbf,1 there exists a path γ5 from tbf,2 to tbf,1. We compose γ5 with γ1 and extend the composition γ1 o γ5 to p. Due to Remark 4.2 this path covers tmj,2. Hence either tmj,1 ≤ tmj,2 or tmj,2 ≤ tmj,1.We exclude the case tmj,2 ≤ tmj,1, because otherwise d(tbf,1, tmj,2) < d(tbf,1, tmj,1). Hence tmj,1 ≤ tmj,2.

Because the partial order ≤ is antisymmetric on EL \ p the two inequalities tmj,2 ≤ tmj,1 and tmj,1 ≤ tmj,2 together imply tmj,2 = tmj,1.



Figure 12 Pairs (tbf, tmj)

Similar one proves, that the net satisfies the check in step 4. Hence the net passes successfully all examinations in step 3,4,5,6,7 and is reduced in step 8. The reduced net is again *or*-well-structured and well-formed by Proposition 4.4, part 3, and the algorithm terminates after finitely many steps.

ii) Assume, that the algorithm terminates for a Boolean loop tree BLT with the output „*or*-well-structured and well-formed“. If we reverse the reduction, then we rebuild BLT successively from its basepoint:

* Step 2: Adjoining a well-formed bp scheme.
* Step 8: Place refinement by N( pi, pt ) = decyl ( cyl ( N( pi, pt ) ) ).

The net cycl ( N( pi, pt ) ) belongs to ELowf, because it results from the possible place refinement of an elementary *or*‑transition by a well-formed bp scheme.

According to Definition 4.1 and Remark 0 the Boolean loop tree BLT is *or*-well-structured and well-formed, QED.

## Branch/fork resolution of or-well-structured Boolean loop trees (Remark)

The above algorithm proceeds by the elimination of *or*-transitions by reducing each well-structured *or*‑alternative to a single place. Afterwards the elementary Boolean loop is a bp scheme and the Genrich-Thiagarajan reduction can be performed.

The same result can be obtained on a second way, too: At first every branch/fork and every merge/join transition is replaced by a subnet of binary transitions according to Proposition 2.10, secondly every elementary binary *or*-alternative is replaced by its branch/fork resolution according to Definition 2.4. Both steps can be gathered into a single one using a recursive branch/fork resolution according to the formula:

x1 or x2 ... xn-1 or xn ⬄ (x1 xor (x2 or x3 ... xn-1 or xn)) xor (x1 and (x2 or x3 ... xn-1 or xn)).

Every numbering of the arcs determines a different interleaving, without changing the net behaviour with respect to bisimulation equivalence.

## Certification of EPCs as well-formed (Remark)

At this point we have reached the final step of our net analysis. Looking back, the whole procedure to certify a given EPC as well-formed comprises the following steps:

* Check the syntax of the EPC according to Definition 1.2.
* Translate the EPC into a Boolean net according to Procedure 1.6.
* Check by standard graph algorithms that the resulting net is a Boolean loop tree.
* Apply algorithm 4.8 to decide, if the Boolean loop tree is *or*-well-structured and well-formed.

The EPC is well-formed iff it passes every step with success.

## Well-formedness of the EPC „Ordering“ (Example)

Figure 13 shows the translation of the EPC „Ordering“ from Figure 4 into a Boolean net according to Procedure 1.6. The transitions K100 and K200 have been introduced in order to connect the basepoint „Start/End“ with the boundary events of the EPC. Nodes without annotation have been introduced by syntactical reasons. Obviously the EPC is well-structured, because the resulting Boolean net is a loop tree without any or-alternatives. Beside the root component with basepoint „Start/End“ there exists a second loop component with articulation point K3.



Figure 13 Ill-formed Boolean net "Ordering"



Figure 14 Well-formed Boolean net "Ordering"

Both loop components are bp schemes, the second - a linear sequence - is obviously well-formed. But the root component is not well-formed. The Genrich-Thiagarajan algorithm reduces the root component to the net from Figure 15 and stops without further reduction to an elementary bp scheme. The problem which hinders well-formedness is the partial synchronisation of the two threads, originating at connector K1 and merging at connector K5. If connector K1 decides to activate place E2 and to deactivate place E3, then the closing *join*-transition K200 gets into deadlock. Even if we change transition K200 to a *merge*-transition, we can produce a similar deadlock by activating place E3 and deactivation place E2.

This problem shows that the attachment of the boundary events of the EPC to the additional place *start/end* requires a careful analysis of the possible combinations of the boundary events: In the present case the process always ends with event E9 and sometimes with event E10 in addition. In order to avoid the partial synchronisation we first duplicate transition F6 and place E9, then join the purchasing alternative at connector K10 and finally synchronise both alternatives by the merge-transition K200, which corresponds to the *branch*-transition K1. The resulting net from Figure 14 is reduced by the Genrich-Thiagarajan algorithm to the well-formed elementary bp scheme from Figure 16. Hence the original EPC „Ordering“ from Figure 4 is well-formed provided we connect the boundary events to the basepoint *start/end* according to the Boolean net of Figure 14.



Figure 15 Genrich-Thiagarajan reduction:   
Termination in the ill-structured case



Figure 16 Genrich-Thiagarajan reduction:   
Termination in the well-structured case

## EPCs and free-choice net systems (Remark)

Every well formed bp scheme can be translated into a live and 1-safe free-choice net system ([GT1984], Theorem 3.13). Hereby one translates the Boolean transitions according to the rules of Table 3 and erases all annotations and all token of logical value *false*. Similarly one can translate the branch/fork-resolution of the Boolean loop tree belonging to a well formed EPC into a live and 1-safe free-choice net system. At the articulations points the free-choice property is garantied by their branching mode according to Definition 3.2.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Bolean loop tree |  |  |  |  |
| Free-choice net |  |  |  |  |

Table 3 Translation of Boolean transitions

# Conclusion and relation to other work

Scheer in joint work with Chen ([CS1994]), as well as other authors ([Brö1996], [LSW1997], [Rod1997]) have proposed translations of EPCs into Petri nets. All these approaches as well as the resulting net classes differ. In addition to his own proposal Rodenhagen ([Rod1997]) compares and comments some of the differences.

Ellis and Nutt ([EN1993]) consider T-nets with logical connectors of type *xor* resp. *and* in order to model the control flow of a process. But they do not mention the connection with bp schemes. After adding the data flow they propose the resulting *Information Control Net* (ICN) as a method to design workflows.

An other approach has been followed by van der Aalst ([Aal1997]). His aim is not the translation of EPCs but - more general - the identification of a class of Petri nets, which is suitable to model the procedures of a workflow. Van der Aalst introduces the class of *sound* *workflow nets*, which proves to be a subclass of well-formed p/t nets. Every loop tree is a sound workflow net. The main difference between the approach of van der Aalst and our approach seems to be the use of different net classes:

* van der Aalst works within the class of p/t nets and introduces the class of workflow nets
* our paper deals with coloured nets and introduces the class of Boolean nets and its subclass of Boolean loop trees.

As noted in Remark 4.12 a Boolean net, which corresponds to a well-formed EPC, can be transformed into a well-formed free-choice net. The resulting free-choice nets form a proper subset of all sound workflow nets.

On the other hand the net analysis of the corresponding EPCs can be made by a reduction algorithm, the problem of the case graph explosion does not appear for EPCs due to the relation between well-formed EPCs and bp schemes. Moreover the solution of the synthesis problem by Genrich and Thiagarajan provides even a complete kit of construction rules for well-formed EPCs.

Bibliography

[Aal1997] *Van der Aalst, Wil M.P.*: Verification of Workflow Nets. In: *Azéma, Pierre; Balbo, Gianfranco* (Eds.): Application and Theory of Petri Nets 1997. Lecture Notes in Computer Science 1248. Springer, Berlin et al. 1997, p. 62-81

[AM1969] *Atiyah, M. F.; Macdonald, I.G.*: Introduction to Commutative Algebra. Addison-Wesley, Reading, Mass. 1969

[BD1990] *Best, Eike; Desel, Jörg*: Partial order behaviour and structure of Petri nets. Formal aspects of computing 2 (1990), p. 123-138

[Brö1996] *Bröker, Axel*: Transformation ereignisgesteuerter Prozeßketten in Prädikat-Transitions-Netze sowie Vergleich der Modellierungstools ARIS-Toolset und INCOME. Diplomarbeit FH Pforzheim, Hochschule für Gestaltung, Technik und Wirtschaft. Pforzheim 1996

[CS1994] *Chen, R.; Scheer, August-Wilhelm*: Modellierung von Prozeßketten mittels Petri‑Netz Theorie. Veröffentlichungen des Instituts für Wirtschaftsinformatik, Heft 107, Saarbrücken 1994

[DE1995] *Desel, Jörg; Esparza, Javier*: Free choice Petri nets. Cambridge University Press, Cambridge 1995

[EN1993] *Ellis Clarence A.; Nutt, Gary J.*: Modelling and Enactment of Workflow Systems. In: *Marsan, Marco Ajmone* (Ed.): Application and Theory of Petri Nets 1993. Lecture Notes in Computer Science 691. Springer, Berlin et al. 1993, p. 1-16

[GT1984] *Genrich, Hartmann; Thiagarajan, Pazhamaneri*: A Theory of Bipolar Synchronisation Schemes. Theoretical Computer Science 30 (1984), p. 241-318

[Jen1992] *Jensen, Kurt*: Coloured Petri Nets. Basis Concepts, Analysis Methods and Practical Use. Springer, Berlin et al. 1992

[KNS1991] *Keller, Gerhard; Nüttgens, Markus; Scheer, August-Wilhelm*: Semantische Prozeßmodellierung auf der Grundlage „Ereignisgesteuerter Prozeßketten (EPK)“. Veröffentlichungen des Instituts für Wirtschaftsinformatik, Heft 89, Saarbrücken 1991

[LSW1997] *Langner, Peter; Schneider, Christoph; Wehler, Joachim*: Ereignisgesteuerte Prozeßketten und Petri-Netze. Universität Hamburg, Fachbereich Informatik, Bericht Nr. 196, FBI-HH-B-196/97, 1997

[Lau1987] *Lautenbach, Kurt*: Linear Algebraic Calculations of Deadlocks and Traps. In: Voss, K., Genrich, H., Rozenberg, G. (Eds.): Concurrency and Nets, p. 315-336. Springer, Berlin et al. 1987

[Nüt1995] *Nüttgens, Markus*: Koordiniert-dezentrales Informationsmanagement: Rahmenkonzept - Koordinationsmodelle - Werkzeug-Shell. Gabler, Wiesbaden 1995

[Rod1997] *Rodenhagen, Jörg*: Darstellung ereignisgesteuerter Prozeßketten (EPK) mit Hilfe von Petrinetzen. Diplomarbeit, Universität Hamburg, Fachbereich Informatik, Hamburg 1997

[SAPa1996] *SAP*: CA R/3 Reference Model. R/3 System Release 3.0. (Product documentation), Walldorf 1996

[SAPb1996] *SAP*: SAP Business Workflow. R/3 System Release 3.0. (Product documentation), Walldorf 1996

[Sch1994] *Scheer, August-Wilhelm*: Business Processes Engineering. Reference Models for Industrial Enterprises. Springer, Berlin et al. 2nd 1994

1. INNOBIS Unternehmensberatung und Software GmbH, Willhoop 7, D-22453 Hamburg, email: P.Langner@innobis.de [↑](#footnote-ref-1)
2. Christoph Schneider Datenerfassung, Paul Robeson Straße 40, D-10439 Berlin, email: 100422.133@compuserve.com [↑](#footnote-ref-2)
3. Softlab GmbH, Zamdorfer Straße 120, D-81677 München, email: wej@softlab.de [↑](#footnote-ref-3)
4. SIMPLE++ is a product of AESOP GmbH, Stuttgart [↑](#footnote-ref-4)
5. ARIS-Toolset is a product of IDS Prof. Scheer GmbH, Saarbrücken [↑](#footnote-ref-5)