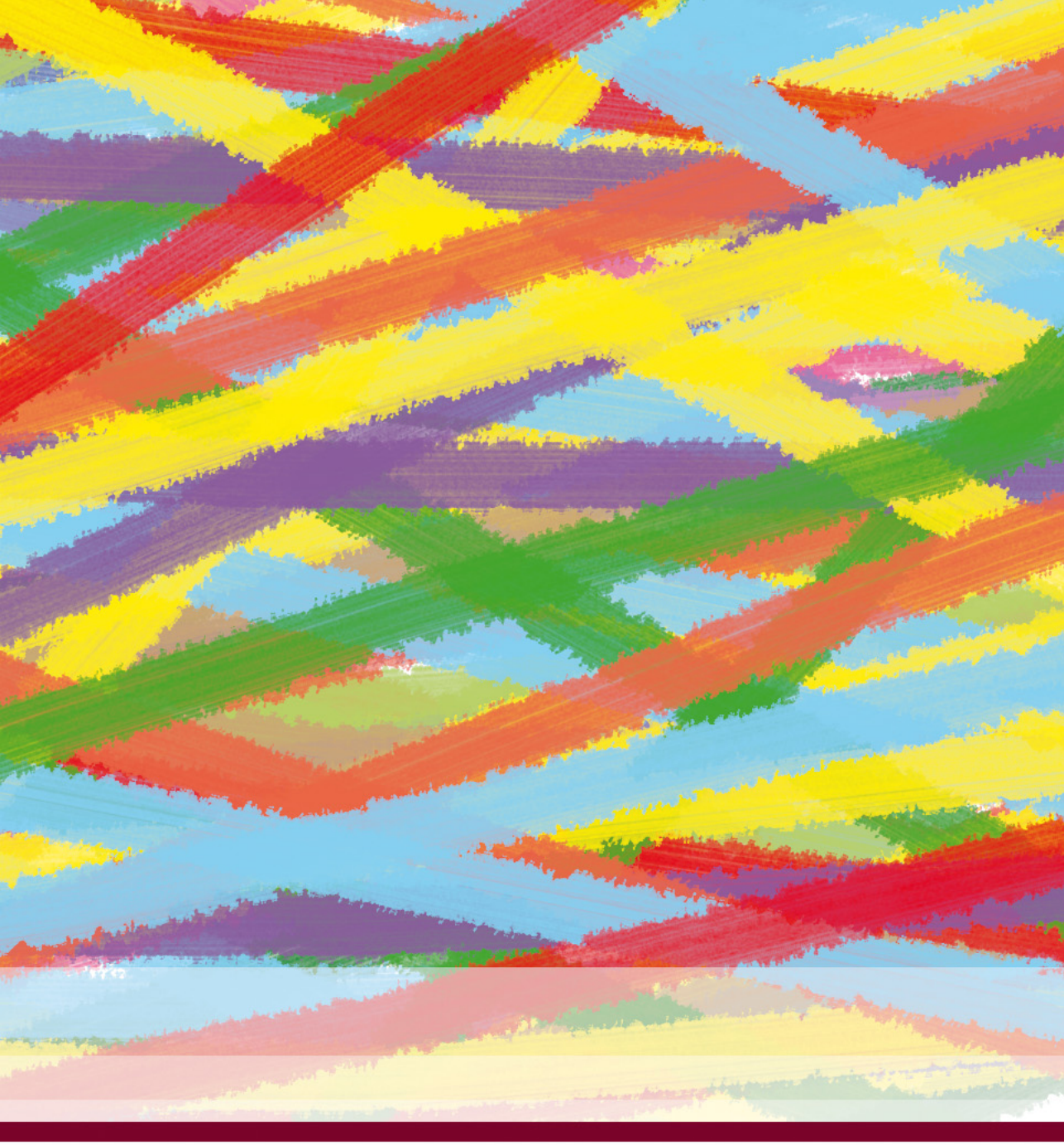


Structuring Process Models

Artem Polyvyanyy

Structuring Process Models

Artem Polyvyanyy

Structuring Process Models

Dissertation

zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften
"doctor rerum naturalium"
— Dr. rer. nat. —
in der Wissenschaftsdisziplin "Praktische Informatik"

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät
der Universität Potsdam

von

Artem Polyvyanyy

geboren am 9. Juli 1983 in Mariupol, Ukraine

Business Process Technology group
Hasso Plattner Institute for Software Systems Engineering
at the University of Potsdam
Potsdam, Germany

January 2012

Abstract

One can fairly adopt the ideas of Donald E. Knuth to conclude that process modeling is both a science and an art. Process modeling does have an aesthetic sense. Similar to composing an opera or writing a novel, process modeling is carried out by humans who undergo creative practices when engineering a process model. Therefore, the very same process can be modeled in a myriad number of ways. Once modeled, processes can be analyzed by employing scientific methods.

Usually, process models are formalized as directed graphs, with nodes representing tasks and decisions, and directed arcs describing temporal constraints between the nodes. Common process definition languages, such as Business Process Model and Notation (BPMN) and Event-driven Process Chain (EPC) allow process analysts to define models with arbitrary complex topologies. The absence of structural constraints supports creativity and productivity, as there is no need to force ideas into a limited amount of available structural patterns. Nevertheless, it is often preferable that models follow certain structural rules.

A well-known structural property of process models is (well-)structuredness. A process model is *(well-)structured* if and only if every node with multiple outgoing arcs (a split) has a corresponding node with multiple incoming arcs (a join), and vice versa, such that the set of nodes between the split and the join induces a single-entry-single-exit (SESE) region; otherwise the process model is *unstructured*. The motivations for well-structured process models are manifold: (i) Well-structured process models are easier to layout for visual representation as their formalizations are planar graphs. (ii) Well-structured process models are easier to comprehend by humans. (iii) Well-structured process models tend to have fewer errors than unstructured ones and it is less probable to introduce new errors when modifying a well-structured process model. (iv) Well-structured process models are better suited for analysis with many existing formal techniques applicable only for well-structured process models. (v) Well-structured process models are better suited for efficient execution and optimization, e.g., when discovering independent regions of a process model that can be executed concurrently.

Consequently, there are process modeling languages that encourage well-structured modeling, e.g., Business Process Execution Language (BPEL) and ADEPT. However, the well-structured process modeling implies some limitations: (i) There exist processes that cannot be formalized as well-structured process models. (ii) There exist processes that when formalized as well-structured process models require a considerable duplication of modeling constructs.

Rather than expecting well-structured modeling from start, we advocate for the absence of structural constraints when modeling. Afterwards, automated methods can suggest, upon request and whenever possible, alternative formalizations that are "better" structured, preferably well-structured. In this thesis, we study the problem of automatically transforming process models into equivalent well-structured models. The developed transformations are performed under a strong notion of behavioral equivalence which preserves concurrency. The findings are implemented in a tool, which is publicly available.

Kurzfassung

Im Sinne der Ideen von Donald E. Knuth ist die Prozessmodellierung sowohl Wissenschaft als auch Kunst. Prozessmodellierung hat immer auch eine ästhetische Dimension. Wie das Komponieren einer Oper oder das Schreiben eines Romans, so stellt auch die Prozessmodellierung einen kreativen Akt eines Individuums dar. Somit kann ein Prozess auf unterschiedlichste Weise modelliert werden. Prozessmodelle können anschließend mit wissenschaftlichen Methoden untersucht werden.

Prozessmodelle liegen im Regelfall als gerichtete Graphen vor. Knoten stellen Aktivitäten und Entscheidungspunkte dar, während gerichtete Kanten die temporalen Abhängigkeiten zwischen den Knoten beschreiben. Gängige Prozessmodellierungssprachen, zum Beispiel die Business Process Model and Notation (BPMN) und Ereignisgesteuerte Prozessketten (EPK), ermöglichen die Erstellung von Modellen mit einer beliebig komplexen Topologie. Es gibt keine strukturellen Einschränkungen, welche die Kreativität oder Produktivität durch eine begrenzte Anzahl von Modellierungsalternativen einschränken würden. Nichtsdestotrotz ist es oft wünschenswert, dass Modelle bestimmte strukturelle Eigenschaften haben.

Ein bekanntes strukturelles Merkmal für Prozessmodelle ist Wohlstrukturiertheit. Ein Prozessmodell ist *wohlstrukturiert* genau dann, wenn jeder Knoten mit mehreren ausgehenden Kanten (ein Split) einen entsprechenden Knoten mit mehreren eingehenden Kanten (einen Join) hat, und umgekehrt, so dass die Knoten welche zwischen dem Split und dem Join liegen eine single-entry-single-exit (SESE) Region bilden. Ist dies nicht der Fall, so ist das Modell *unstrukturiert*. Wohlstrukturiertheit ist aufgrund einer Vielzahl von Gründen wünschenswert: (i) Wohlstrukturierte Modelle sind einfacher auszurichten, wenn sie visualisiert werden, da sie planaren Graphen entsprechen. (ii) Wohlstrukturierte Modelle zeichnen sich durch eine höhere Verständlichkeit aus. (iii) Wohlstrukturierte Modelle haben oft weniger Fehler als unstrukturierte Modelle. Auch ist die Wahrscheinlichkeit fehlerhafter Änderungen größer, wenn Modelle unstrukturiert sind. (iv) Wohlstrukturierte Modelle eignen sich besser für die formale Analyse, da viele Techniken nur für wohlstrukturierte Modelle anwendbar sind. (v) Wohlstrukturierte Modelle sind eher für die effiziente Ausführung und Optimierung geeignet, z.B. wenn unabhängige Regionen eines Prozesses für die parallele Ausführung identifiziert werden.

Folglich gibt es eine Reihe von Prozessmodellierungssprachen, z.B. die Business Process Execution Language (BPEL) und ADEPT, welche den Modellierer anhalten nur wohlstrukturierte Modelle zu erstellen. Solch wohlstrukturiertes Modellieren impliziert jedoch gewisse Einschränkungen: (i) Es gibt Prozesse, welche nicht mittels wohlstrukturierten Prozessmodellen dargestellt werden können. (ii) Es gibt Prozesse, für welche die wohlstrukturierte Modellierung mit einer erheblichen Vervielfältigung von Modellierungskonstrukten einhergeht.

Aus diesem Grund vertritt diese Arbeit den Standpunkt, dass ohne strukturelle Einschränkungen modelliert werden sollte, anstatt Wohlstrukturiertheit von Beginn an zu verlangen. Anschließend können, sofern gewünscht und wo immer es möglich ist, automatische Methoden Modellierungsalternativen vorschlagen, welche “besser” strukturiert sind, im Idealfall sogar wohlstrukturiert. Die vorliegende Arbeit widmet sich dem Problem der automatischen Transformation von Prozessmodellen in verhaltensäquivalente wohlstrukturierte Prozessmodelle. Die vorgestellten Transformationen erhalten ein strenges Verhaltensequivalenzkriterium, welches die Parallelität wahrt. Die Resultate sind in einem frei verfügbaren Forschungsprototyp implementiert worden.

... structuring process ... is no substitute for good design. The transforms ... might help to unravel some knotty problems, but they cannot produce logical poetry from tangled nonsense.

(G.Oulsnam)

Contents

Part I.

Modeling Behavior

1. Introduction

In the introductory chapter of this thesis we discuss the notion of a behavioral model and the principles of behavioral modeling (see in Section 1.1), talk about structurally constrained behavioral models as well as about their advantages and disadvantages (see in Section 1.2), formulate the structuring problem (refer to Section 1.3), hint at the results achieved in this thesis (refer to Section 1.4), and inform the reader on the outline of this thesis (see in Section 1.5).

Behavioral models describe dynamic aspects of real world or designed systems. In this thesis, we study the problem of automatically transforming behavioral models into equivalent and structurally constrained behavioral models, viz. well-structured behavioral models. The solution to this problem, which will be proposed in the subsequent chapters of the thesis, allows for preserving the concurrency of an original behavioral model in the newly constructed well-structured behavioral model, which is of particular interest for various applications.

1.1. Behavioral Models

Modeling is at the core of many engineering disciplines. Models are developed to cope with the complexity of real world phenomena and are abstractions thereof [3]. A *model* is a reduced, but a sufficient, representation of a phenomenon which is suitable for a particular purpose, e.g., simulating a *system* prior to starting with its implementation. A *conceptual model* represents entities and relations between entities from a certain domain [44]. Conceptual models aim at explaining ambiguous terms from the domain of interest and finding correct relations between entities. By developing conceptual models, engineers gain common understanding, and hence define common playground, for iterating solutions within the problem domain. Conceptual models are widely used for analysis, verification, simulation, and communication purposes. Useful models contain a sufficient amount of information for solving the envisioned engineering problem.

In this thesis, we study conceptual models of a particular kind, viz. behavioral models. A *behavioral model* describes dynamic aspects of a real world or a designed system. The main building bricks of behavioral models are entities like *events* (phenomena located at single points in time) and *tasks* (pieces of work performed within certain periods of time), as well as ordering relations between events and tasks on the time axis. In other words, we study models that describe *behaviors* which can be perceived as partial orders of entities, such that ordered entities can only occur in sequence, whereas executions of unordered entities may overlap in time. Computer programs [70, 51], service compositions [35, 72, 102], and (business) process models [155, 27, 135] are examples of behavioral models.

Different modeling paradigms suggest different styles for formalizing behaviors. The choice of a modeling paradigm for solving a problem is usually carried out based on characteristics of the problem at hand; note that a paradigm is usually most suitable when solving problems of a particular type. Over the last decades, the imperative paradigm has shown its feasibility when solving a wide range of engineering problems which are concerned with formalizing behaviors. Within the imperative paradigm, designers of behavioral models specify *how* the system, which is described in the model, should be able to achieve the result by providing precise instructions in the form of potential execution sequences that are composed of events and tasks. As for the other modeling styles, such as the declarative [133, 134, 92, 103] or data-centric [17, 80, 78] design methodology, one can usually observe that those find their use and are beneficial only for solving very specific types of problems.

In this thesis, we study behavioral models developed using imperative languages, i.e., languages which follow the imperative modeling paradigm. There exist many imperative modeling languages. It is often the case that behavioral models, which are developed by employing imperative languages, can be encoded as directed annotated graphs, or *executable graphs*. In executable graphs, nodes represent tasks or events and directed edges capture execution ordering constraints between the nodes, i.e., the target of an edge may be executed once the source is accomplished.

To conclude, in this thesis, we study behavioral models that are specified by following the imperative modeling paradigm and which can be formalized in the form of executable graphs.

1.2. Well-structured Modeling

On the previous page, we reduced the scope of this thesis to studies of behavioral models which follow the imperative modeling paradigm. We also stated that behavioral models that are described by employing imperative languages can be formalized as directed graphs such that nodes of a graph represent entities from the problem domain, and directed edges encode causal dependencies between adjacent nodes. This observation provides the opportunity for performing structural investigations on behavioral models by analyzing the structural characteristics of the underlying executable graphs.

State-of-the-art languages for describing behavioral models allow models to have almost any topology, e.g., Business Process Model and Notation (BPMN) [2] or Event-driven Process Chains (EPC) [64]. However, it is often preferable that behavioral models follow some structural rules. A well-known property of behavioral models is that of *(well-)structuredness* [67]:

> *A behavioral model is **(well-)structured**, if and only if every node with multiple outgoing arcs (a split) has a corresponding node with multiple incoming arcs (a join), and vice versa, such that the part of the behavioral model between the split and the join forms a single-entry-single-exit (SESE) component; otherwise the model is **unstructured**.*

Figure 1.1 shows two behavioral models captured using the BPMN language. An execution of the behavioral model in Figure 1.1(a) starts at node i, which marks the creation of a new *instance* of the model. The thread of control is then immediately passed to node u where a decision is carried out on how to proceed, either by executing "*Pay by check*" task or by executing "*Pay by cash*" task. Once the selected task is accomplished, the thread of control is passed either to node v or to node w, respectively. Splits v and w introduce concurrency to the execution, i.e., once the execution reaches either of these nodes the tread of control is replicated along every outgoing edge of the node. Joins x and y immediately pass on every incoming thread of control to the only outgoing edge. Therefore, executions of tasks "*Approve*" and "*Update account*" may overlap in time. Once both tasks are accomplished, the execution is synchronized at join z. The execution of an instance terminates once the thread of control reaches node o.

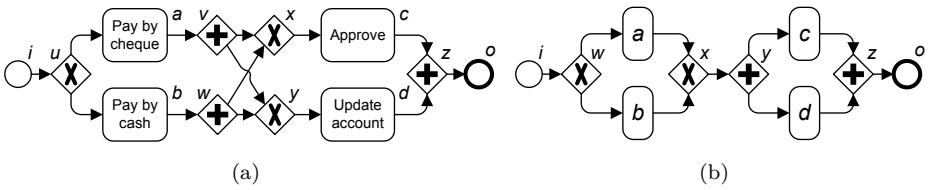


Figure 1.1. Behavioral models: (a) unstructured, and (b) well-structured

The behavioral model in Figure 1.1(a) is unstructured. In the model, split u has corresponding join z; together they define a SESE component with entry u

and exit z. Yet, splits v and w have no corresponding joins. Similarly, joins x and y have no corresponding splits. Figure 1.1(b) shows a well-structured behavioral model, which in fact is equivalent to the behavioral model in Figure 1.1(a), i.e., both models describe the same behavior. Every split of the model in Figure 1.1(b) has a corresponding join, e.g., split w has corresponding join x and split y has corresponding join z. Each of these corresponding pairs defines a SESE component, where split is an entry and join is an exit of the SESE component. Similarly, joins x and z have their corresponding splits. Note that Figure 1.1(b) uses short-names for tasks $(a, b, c, \ldots)$, which appear next to each task in Figure 1.1(a).

Structured behavioral models have many advantages over unstructured ones:

- Well-structured models are easier to layout [28, 69, 121]. Well-structured behavioral models are captured by planar graphs and can hence be drawn on the plane in such a way that no edges cross each other, which is one of the most important aesthetics when drawing behavioral models. Among other aesthetics that can be easily fulfilled when drawing well-structured behavioral models are minimization of drawing area, minimization of number of overlapping elements, maximization of edges which are drawn one after another in the prescribed direction, orthogonal drawing of edges, i.e., as a sequence of horizontal and vertical line segments, etc.
- It has been empirically shown that well-structured behavioral models are easier to comprehend by humans and tend to have fewer errors than unstructured ones. In [76, 86], the findings strongly support the importance of well-structuredness for the quality of behavioral models.
- By transforming unstructured behavioral models into well-structured ones, one extends the applicability of analysis techniques which are only applicable for well-structured models [75], and improves translations between models captured in different languages [81, 98, 154]. Many analysis techniques for behavioral models, e.g., aggregate Quality of Service computation for composite services, can be defined for well-structured models in a straightforward manner [26, 59, 60, 15, 94, 16, 57, 58]. Therefore, a technique for translating models with unstructured parts into well-structured ones allows for a separation of concerns.
- Well-structured behavioral models are better suited for execution optimization [43, 56, 100, 73], i.e., they are better suited for generating executable code for parallel machines and for partitioning behavioral models on multiprocessor machines; one can achieve better reduction in communication and synchronization costs.
- Well-structured behavioral models are favored in the context of refactoring large model repositories [24, 125, 152]. Refactorings within model repositories can improve the management of model complexity by making models both easier to understand and to maintain. For instance, the technique for detecting and refactoring clones [125], i.e., identical parts of behavioral models in different models of the repository, can discover clones faster if applied to well-structured models.
- Etc.

The arguments listed above lead to a provocative question: If well-structured behavioral models are so good, why do unstructured models exist? One approach to modeling behavior can be to forbid unstructuredness on the syntactical level and to thus ensure that all models are good, i.e., well-structured. The languages which advocate well-structured modeling are, for example, Business Process Execution Language (BPEL) [1] and ADEPT [117, 118]. However, a modeling methodology that confines itself to "well-structured" languages faces certain limitations:

- There exist behavioral models with concurrency that have no equivalent well-structured version [68]. This simply means that certain behaviors cannot be modeled.
- Well-structured modeling implies design time constraints and thus limits creativity and lowers productivity of model designers [55, 119]. Designers should try hard to produce well-structured models, but they must be able to introduce unstructured parts when those are required.

The discussion of whether to promote well-structured modeling or to allow unstructuredness can be projected onto the problem of elimination of Go To statements in computer programs, i.e., Go To statements can be seen as the source of unstructuredness in programs. Despite decades of debates [25, 55, 159, 93, 119], Go To statements are still present in state-of-the-art high-level programming languages (even though it has been formally shown that Go To statements are unnecessary [13]). If one shifts from sequential reality to the reality where things can happen in parallel, unstructuredness is in the inherent nature of models [68]. As a general observation, one can conclude that well-structured behavioral models are less expressive than unstructured ones.

Taking into consideration all of the above, we advocate the absence of structural limitations when modeling behavior. Methodology which supports unstructuredness allows for a large degree of creativity when modeling; note that behavioral models are primarily developed by humans who undergo creative practices when solving complex engineering problems. Alternatively, unstructured behavioral models often result from model synthesis techniques, such as process mining [137]. Given an unstructured behavioral model, scientific methods can propose, upon request and whenever possible, alternative formalizations that are "better" structured, preferably well-structured. Therefore, one should be allowed to specify the behavioral model as in Figure 1.1(a) and, if requested, the equivalent well-structured behavioral model, as in Figure 1.1(b), should be constructed automatically. If one possesses a technique which allows the construction of a well-structured version of an unstructured behavioral model, one obtains the most benefits of unstructured modeling and well-structured analysis.

1.3. The Structuring Problem

This section formulates the structuring problem. More precisely, we define the family of structuring problems and point out one particular instance of the problem; it is this instance for which we shall search for a solution in the thesis at hand.

Figure 1.2 visualizes the overall setting of the structuring problem. In the figure, the oval region represents the set of all behavioral models; every dot inside the region represents a behavioral model (there are infinitely many behavioral models). For instance, dots p_1, p_2, and p_3 represent three distinct behavioral models. The very same behavior can be represented by several different models, which gives a rise for a behavioral equivalence relation. A behavioral equivalence relation is an equivalence relation on the set of all behavioral models. A behavioral equivalence relation partitions the set of all models into equivalence classes such that every model is in one and only one equivalence class of the partition. Two models describe the same behavior if and only if they are elements of the same equivalence class. In the figure, C_1, C_2, and C_3 are equivalence classes (induced by some behavioral equivalence relation) which contain models p_1, p_2, and p_3, respectively (there are infinitely many equivalence classes). One can conclude from Figure 1.2 that models p_1, p_2, and p_3 describe different behaviors.

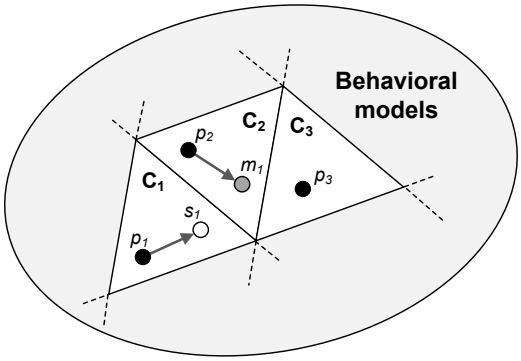


Figure 1.2. Behavioral equivalence relation on the set of all behavioral models and its three equivalence classes

Given behavioral model p and a behavioral equivalence relation, the structuring problem can be formulated in the following question:

> *Does the equivalence class, the one which contains behavioral model p and is induced by the behavioral equivalence relation, contain behavioral model q which is "better" structured than p (preferably well-structured)?*

The solution to the structuring problem, i.e., the answer to the question above, can be implemented either by constructing q from p, or by providing the evidence that q does not exist. Given a behavioral model as input, the solution to its structuring problem might result in several outcomes:

- There exists a behavioral model which is well-structured and describes the same behavior as the given behavioral model p_1. In Figure 1.2, we visualize this situation with dot s_1 inside equivalence class C_1; dot s_1 represents a well-structured behavioral model which captures the same behavior as model p_1. Note that s_1 might be equal to p_1 if p_1 is well-structured.

- There exists no behavioral model which is well-structured and describes the same behavior as the given behavioral model p_2, but there exists a behavioral model which is "better" structured than p_2 and describes the same behavior as p_2. In the figure, we visualize this situation with dot m_1 inside equivalence class C_2. Model m_1 is not well-structured, but it exhibits more structural information than model p_2. Moreover, there exists no model in C_2 which is "better" structured than m_1. We refer to model m_1 as the *maximally-structured* version of p_2. Note that m_1 might be equal to p_2 if p_2 is maximally-structured.
- There exists no behavioral model which is well-structured (or maximally-structured) and describes the same behavior as the given behavioral model p_3. In the figure, we visualize this situation with equivalence class C_3. We refer to model p_3 as *inherently unstructured*.

The arcs from p_1 to s_1 and from p_2 to m_1 in Figure 1.2 represent construction paths (algorithms) which, given a behavioral model, construct its structured version; note that a construction path might as well be empty.

The structuring problem can be instantiated using different behavioral models and different behavioral equivalence relations. The expressive power and the semantics of different languages used to capture behavioral models most probably require different approaches when solving the structuring problem. The choice of a concrete behavioral model defines the oval region in Figure 1.2 – the space of the structuring problem, whereas the choice of a concrete behavioral equivalence relation – equivalence classes on the set of all behavioral models.

In this thesis, we concretize the structuring problem with the notion of a *process model*, which can be seen as a greatest common divisor of common languages for defining behavioral models with concurrency. Furthermore, we concretize the structuring problem with a behavioral equivalence relation which requires the preservation of concurrency in equivalent models, viz. *fully concurrent bisimulation*. Such a configuration of the structuring problem is of a particular interest for many use cases. We shall provide the motivation and discuss related work for this particular configuration of the structuring problem later in Chapter 5, and provide the solution for this instance of the problem in Chapter 6.

In this thesis, our focus is on the control flow perspective of the structuring problem. State-of-the-art languages for describing behavioral models (mainly those that originate in the domain of business process management [155]) take a broader view by providing means for specifying data flow perspective, resource perspective, organizational view, etc., as parts of behavioral models. The primary purpose of behavioral models is to orchestrate the execution of tasks on the time axis. We believe that in future works, the solution to the structuring problem – as proposed in this thesis – can be naturally generalized to incorporate other perspectives of behavioral modeling.

1.4. Results of this Thesis

This monograph contributes a novel technique for structuring behavioral models. The original unstructured behavioral model and its newly constructed equivalent

well-structured version have the same semantics and thus describe the same behavior, also in respect to potential concurrent executions of tasks. Therefore, for instance, given the behavioral model in Figure 1.1(a), we are able to construct its well-structured version shown in Figure 1.1(b).

The solution is engineered by re-using and further specifying several results from different fields. The solution rests on techniques for graph parsing, results from the theory of two-structures, results on Petri nets and net unfoldings, etc. On the way towards a solution to the structuring problem we contribute to the technique for parsing behavioral models into hierarchies of SESE components, develop an approach for performing abstractions within behavioral models, specify a criterion for truncating net unfoldings (which is of particular interest for the structuring problem), define the notion of an ordering relations graph (a convenient way for capturing behavior), etc. The lessons we have learned when working on the solution to the structuring problem allowed us to specify a structural approach for the verification of behavioral models and to propose a framework for organizing structural investigations on behavioral models.

For a more elaborate discussion of the main contributions of this thesis please refer to Section 9.1.

1.5. Structure of this Thesis

In the concluding section of this chapter, we describe the overall structure of the thesis. The thesis consists of four parts. The order in which content is proposed to the reader is optimized for the gradual presentation of the main story on structuring of behavioral models. However, certain chapters of the thesis can be of interest even when addressed in isolation. In the following, we explain the dependencies between different chapters of this thesis.

Part I: Modeling Behavior

The first part consists of two chapters: Chapter 1 is devoted to a discussion of common approaches to modeling behavior. In this chapter, we have already stressed the role of structural constraints when modeling behavior, see Section 1.2, and formulated the structuring problem, see Section 1.3. We shall propose a solution to one particular instance of the structuring problem in Part III of the thesis. Chapter 2 introduces basic notions which will be used within the thesis to convey the findings. Importantly, in Chapter 2, we introduce the notion of a *process model* – a simplistic, yet sufficient, language for specifying behavioral models; it is this language that we shall use in the subsequent parts of this thesis when dealing with the structuring of behavioral models.

Part II: Parsing and Abstraction

The second part consists of two chapters: Chapter 3 presents a technique for parsing behavioral models, viz. the Refined Process Structure Tree. Chapter 4 then uses the parsing technique to define a technique for abstracting behavioral models, viz.

the triconnected abstraction. Both techniques exploit structural characteristics of behavioral models. The parsing technique allows for decompositions of behavioral models into hierarchies of SESE components. The components can be treated as self-contained units of behavior in the models. The triconnected abstraction technique proposes to employ components obtained during parsing of behavioral models in order to bring the models to higher abstraction levels by neglecting insignificant details within the components. Both the parsing and abstraction technique are employed in Part III to modularize the structuring problem and to concentrate on the essences of structuring.

Part III: Structuring

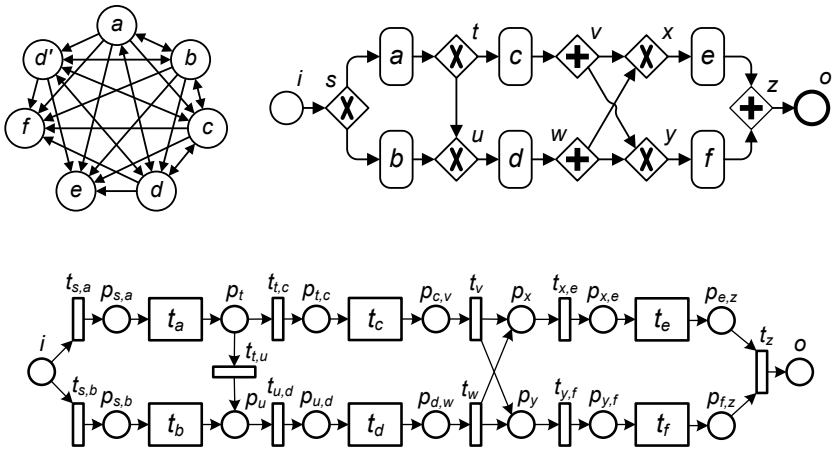
The third part consists of two chapters: Prior to proceeding with the presentation of different techniques for structuring process models, Chapter 5 sets the stage by giving the foundations. In Chapter 5, we formally define the notion of a well-structured process model based on its *parse tree* (the parse tree is discussed in Part II, Chapter 3), elaborate on a behavioral equivalence notion we intend to preserve when constructing well-structured versions of unstructured models, and give the still missing preliminaries. Chapter 6 is devoted to the presentation of structuring techniques, which constitute the main contribution of this thesis. All the structuring techniques rely on the basic technique for structuring acyclic process models. Further structuring techniques address the particular cases of structuring process models with multiple source and/or multiple sink nodes, maximal-structuring of acyclic process models, and structuring of process models with arbitrary cyclic paths.

Part IV: Analysis

The fourth and the last part of this thesis also consists of two chapters: Chapter 7 generalizes the principles for parsing behavioral models from Chapter 3 and proposes a stepwise technique for structural verification of behavioral models. As behavioral models that we allow for structuring must be correct, one can employ the verification technique from this chapter to check the correctness of models prior to structuring them. Chapter 8 reuses the experience gained in the course of this thesis and proposes a framework for organizing structural investigations on behavioral models. We believe that the framework will find its use in many use cases which deal with the structural analysis of behavioral models.

Finally, Chapter 9 concludes the thesis at hand. The chapter summarizes the main contributions of the thesis, lists open problems, and states research opportunities for future work.

2. Basic Notions



This chapter presents some well-established notions and formalisms which will be used later to convey the findings. The corresponding definitions and formal notations are discussed to the extent necessary. Section 2.1 opens the chapter with basics on graph theory. Then, Section 2.2 discusses the notion of a two-structure, which is a generalization of the notion of a directed graph. Afterwards, we present formalisms for describing dynamic behavior: Section 2.3 looks at Petri nets, a well-known formalism for modeling distributed systems. Section 2.4 is devoted to workflow nets, a structural subclass of Petri nets designed to capture workflow procedures. Finally, Section 2.5 discusses process models, which are simplistic, yet sufficient, behavioral models for addressing the structuring problem.

2.1. Graphs

Graphs are mathematical structures used to model pairwise relations between elements of a certain collection. In this section, we present the basics of graphs. We give a small number of well-known definitions from the graph theory, cf., [14, 46], which fulfill our needs for the subsequent sections. Section 2.1.1 present undirected, directed, and multi-graphs, whereas Section 2.1.2 discusses adjacency matrices – an approach to formalize graphs.

2.1.1. Undirected, Multi-, and Directed Graphs

Let V be a nonempty finite set of elements, and denote by

$$\mathcal{E}_2(V) = \{\{v_1, v_2\} \mid v_1, v_2 \in V,\ v_1 \neq v_2\}$$

the set of all subsets of V of two distinct elements.

Definition 2.1 (Graph).
An ordered pair $G = (V, E)$, where $E \subseteq \mathcal{E}_2(V)$, is called a *graph*. ⌟

The elements of V are called *vertices*, and those of E are called *edges* of the graph. The vertex set of a graph G is denoted by V_G and its edge set by E_G. Note that in the following we omit subscripts of vertex sets and edge sets where the context is clear. Vertices and edges of a graph can also be referred to as *nodes* and, respectively, *arcs* of the graph. A graph G can be visualized as a plane figure where each vertex is shown as a circle while each edge is drawn as a line segment (or a curve segment) which connects the vertices of an edge. Figure 2.1(a) is a drawing of graph G with vertices $V_G = \{v_1, v_2, v_3, v_4\}$ and edges $E_G = \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_2, v_3\}, \{v_3, v_4\}\}$.

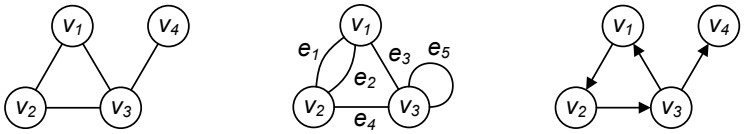


Figure 2.1. (a) A graph, (b) a multi-graph, and (c) a directed graph

Graphs can be generalized to *multi-graphs* by allowing *self-loops* and multiple edges between vertices; a *self-loop* is an edge that connects a vertex with itself, e.g., an edge $\{v, v\}$, $v \in V_G$ in graph G.

Definition 2.2 (Multi-graph).
An ordered triple $M = (V, E, \ell)$, where $E = \{e_1, e_2, \ldots, e_n\}$, $n \in \mathbb{N}_0$, is a set of edges and $\ell : E \to \mathcal{E}_2(V) \cup \{\{v, v\} \mid v \in V\}$ is a function that maps every edge to a pair of vertices, is called a *multi-graph*. ⌟

$\mathbb{N}_0$ is the set of natural numbers including zero. Figure 2.1(b) is a visualization of the multi-graph M with vertices $V_M = \{v_1, v_2, v_3\}$, edges $E_M = \{e_1, e_2, e_3, e_4, e_5\}$, and function ℓ, such that $\ell(e_1) = \{v_1, v_2\}$, $\ell(e_2) = \{v_1, v_2\}$, $\ell(e_3) = \{v_1, v_3\}$, $\ell(e_4) = \{v_2, v_3\}$, and $\ell(e_5) = \{v_3, v_3\}$.

We assume that the mapping ℓ is fixed, so that a *subgraph* can be identified with a pair (V', E'), where $V' \subseteq V$ and $E' \subseteq E$ such that each edge in E' connects only nodes in V'. Let $F \subseteq E$ be a set of edges, $G_F = (V_F, F)$ is the subgraph *formed by* F if V_F is the smallest set of nodes such that (V_F, F) is a subgraph. For instance, $(\{v_1, v_2\}, \{e_1, e_2\})$ is a subgraph of the graph in Figure 2.1(b); this one is formed by the set of edges $\{e_1, e_2\}$.

So far we have looked at graphs composed of unordered edges, such graphs are also referred to as *undirected* graphs. A graph $D = (V, E)$ is *directed* if the edges are ordered. Again, let V be a nonempty finite set of vertices, and denote by

$$E_2(V) = \{(v_1, v_2) \mid v_1, v_2 \in V,\ v_1 \neq v_2\}$$

the set of all ordered pairs of V of two distinct vertices.

Definition 2.3 (Directed graph).
An ordered pair $G = (V, E)$, where $E \subseteq E_2(V)$, is called a *directed* graph. ⌟

Figure 2.1(c) shows the directed graph D with vertices $V_D = \{v_1, v_2, v_3, v_4\}$ and directed edges $E_D = \{(v_1, v_2), (v_2, v_3), (v_3, v_1), (v_3, v_4)\}$. Similar to the undirected case, one can talk about *directed multi-graphs* if one allows self-loops and multiple edges between vertices in directed graphs. For a directed edge $e = (v_1, v_2) \in E_2(V)$, its *reverse*, denoted by e^{-1}, is the edge (v_2, v_1).

A node $v_1 \in V_G$ of a graph G is said to be *adjacent* to another node $v_2 \in V_G$ of G, if v_1 and v_2 are connected with an edge in G. We also say that a node $v_1 \in V_G$ of a graph G is *incident* with an edge of G, if the edge connects v_1 with some vertex of G. Finally, two edges e_1 and e_2 of a graph G are *adjacent* if they share the same vertex.

2.1.2. Adjacency Matrix Representation of Graphs

In the previous section, we showed that graphs can be formalized as sets of elements or drawings. Additionally, it is a common practice to encode graphs as *adjacency matrices*, or *adjacency arrays*. An adjacency matrix provides means for defining which vertices of a graph are adjacent to which other vertices. The *adjacency matrix* representation of a graph $G = (V, E)$ is a coloring of the set $\mathcal{E}_2(V)$ with two colors, e.g., 0 and 1, where 0 indicates the absence and 1 the presence of the corresponding edge in the graph. Therefore, an adjacency matrix of a graph G can be given by a characteristic function $\mathcal{I}_G : \mathcal{E}_2(V) \to \{0, 1\}$. Table 2.1(a) encodes the adjacency matrix of the graph in Figure 2.1(a). The table specifies $\mathcal{I}_G(\{v_1, v_2\}) = \mathcal{I}_G(\{v_1, v_3\}) = \mathcal{I}_G(\{v_2, v_3\}) = \mathcal{I}_G(\{v_3, v_4\}) = 1$ and $\mathcal{I}_G(\{v_1, v_4\}) = \mathcal{I}_G(\{v_2, v_4\}) = 0$.

Similarly, the adjacency matrix of a directed graph D can be given by an characteristic function $\mathcal{I}_D : E_2(V) \to \{0, 1\}$. Table 2.1(b) specifies the adjacency matrix of the directed graph in Figure 2.1(c); the matrix specifies $\mathcal{I}_D((v_1, v_2)) = \mathcal{I}_D((v_2, v_3)) = \mathcal{I}_D((v_3, v_1)) = \mathcal{I}_D((v_3, v_4)) = 1$ and 0 for all other edges in $E_2(V)$.

(a)

	v_1	v_2	v_3	v_4
v_1		1	1	0
v_2	1		1	0
v_3	1	1		1
v_4	0	0	1	

(b)

	v_1	v_2	v_3	v_4
v_1		1	0	0
v_2	0		1	0
v_3	1	0		1
v_4	0	0	0	

Table 2.1. Adjacency matrix representations of: (a) the graph in Figure 2.1(a) and (b) the directed graph in Figure 2.1(c)

2.2. Two-Structures

This section presents basic notions from the theory of two-structures [31, 32, 29, 30]. *Two-structures*, or *2-structures*, relate to graphs in two ways. Every graph can be represented as a two-structure while every two-structure defines a family of graphs. Section 2.2.1 presents basic definitions, whereas Section 2.2.2 discusses the class of reversible two-structures.

2.2.1. Definition of a Two-Structure

The notion of a *two-structure* is a generalization of the notion of a directed graph [31]. The adjacency matrix representation of a directed graph $D = (V, E)$ is a coloring of the set $E_2(V)$ with two colors, see Section 2.1.2. A two-structure allows an arbitrary coloring of the set $E_2(V)$.

Definition 2.4 (Two-structure).
An ordered pair $S = (N, R)$, where N is a nonempty finite set of *nodes*, or *domain*, and R is an equivalence relation on $E_2(N)$, is called a *two-structure*. ⌟

We use $dom(S)$ and $rel(S)$ to denote N and R, respectively. We say that two edges $e_1, e_2 \in E_2(N)$ are *equivalent* iff $e_1 \; R \; e_2$. For an edge $e \in E_2(N)$, we denote by $eR = \{e' \mid e \; R \; e'\}$ an equivalence class of R that contains edge e. We also refer to eR as the *edge class* of e.

A two-structure can be seen as a complete directed graph with labeled (colored) edges, where $\alpha : E_2(N) \to C$ is a coloring function corresponding to the edge classes, such that $e_1 \; R \; e_2$ iff $\alpha(e_1) = \alpha(e_2)$; C is a set of colors. Observe that a coloring function α is not unique as the choice of colors can be arbitrary. We say

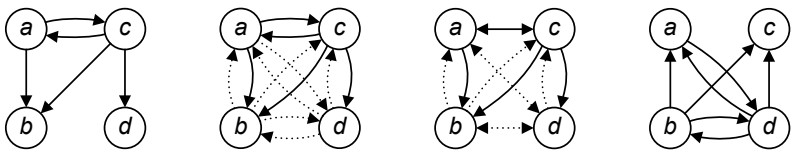


Figure 2.2. (a),(d) Directed graphs, and (b),(c) two-structures

that an edge $e \in E_2(N)$ is *symmetric* if and only if $\alpha(e) = \alpha(e^{-1})$; otherwise e is *asymmetric*. A two-structure is *symmetric* if and only if all its edges are symmetric. We call it *asymmetric* if and only if all its edges are asymmetric.

Figure 2.2(a) shows the directed graph $D = (V, E)$ with vertices $V = \{a, b, c, d\}$ and edges $E = \{(a, c), (c, a), (a, b), (c, b), (c, d)\}$. Figure 2.2(b) presents one of the possible corresponding two-structures $S = (V, R)$ of D. The domain of the two-structure is composed of vertices of D, whereas the equivalence relation R defines two equivalence classes of edges: one class contains edges E (drawn with solid lines) and the other one contains edges $E_2(V) \setminus E$ (drawn with dotted lines). Figure 2.2(c) shows the same two-structure using a simplified notation, i.e., symmetric edges are drawn as two-sided arrows. Please observe that the correspondence between the graph and the two-structure is rather arbitrary, as one can also accept the two-structure as corresponding to the graph in Figure 2.2(d) by exchanging the roles of its equivalence classes. Alternatively, one can define the correspondence between a graph and a two-structure by using larger sets of colors.

2.2.2. Reversibility of Two-Structures

An important subclass of two-structures is the class of *reversible* two-structures.

Definition 2.5 (Reversible two-structure).
A *two-structure* $S = (N, R)$ is *reversible* iff for every pair of edges $e_1, e_2 \in E_2(N)$ holds if e_1 and e_2 are equivalent, then edges e_1^{-1} and e_2^{-1} are also equivalent, i.e., $\forall\ e_1, e_2 \in E_2(N) : e_1\ R\ e_2\ \Rightarrow\ e_1^{-1}\ R\ e_2^{-1}$. ⌟

The next construction allows one to often consider reversible, rather than arbitrary, two-structures. Given an arbitrary two-structure, one can always construct its corresponding *reversible version* by employing the *reversible refinement*.

Definition 2.6 (Reversible refinement, Reversible version).
Let $S = (N, R)$ be a two-structure.

- The *reversible refinement* of R, denoted by $R^\star$, is defined by: for all $e_1, e_2 \in E_2(N)$, $e_1\ R^\star\ e_2$ iff $e_1\ R\ e_2$ and $e_1^{-1}\ R\ e_2^{-1}$.
- The *reversible version* of S, denoted $S^\star$, is the two-structure $(N, R^\star)$. ⌟

Figure 2.3(a) shows the reversible version of the two-structure in Figure 2.2(c). The reversible two-structure has four equivalence classes, which are visualized by lines of different types. The two-structure can be formalized by a coloring function which maps the set of all edges on the set of four colors. Table 2.2 encodes one

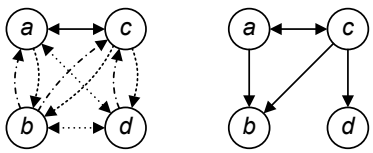


Figure 2.3. Two drawings of a two-structure

	a	b	c	d
a		2	1	0
b	3		3	0
c	1	2		2
d	0	0	3	

Table 2.2. A coloring

possible coloring function which is a mapping on the set of colors $C = \{0, 1, 2, 3\}$. In Figure 2.3(a), the dotted lines encode edges of color 0, solid lines represent edges of color 1, dashed lines encode edges of color 2, and dash-dotted lines define edges of color 3.

The drawing of reversible two-structures can be greatly simplified compared to the drawing of general two-structures by: (i) omitting reverse edges of the edges from the chosen edge classes, (ii) drawing a line segment or a two-sided arrow to encode a symmetric edge, and (iii) omitting one symmetric edge class [29]. Therefore, the simplified drawing of the two-structure in Figure 2.3(a) can be similar to the drawing of the graph in Figure 2.2(a), see Figure 2.3(b). Due to a design decision, in Figure 2.3(b), we omitted edges of the class 3 and edges of the symmetric class 0, whereas edges of the class 2 are drawn as solid lines.

2.3. Petri Nets

Petri nets are a well-known formalism for modeling distributed systems. Petri nets are the means to define the structure of distributed systems. The dynamic behavior of a Petri net, and hence of the corresponding distributed system, is captured by the token game which takes place directly in the net. This section is devoted to Petri nets, their execution semantics and properties.

In Section 2.3.1, we give the definition of a Petri net along with some basic supporting concepts. Section 2.3.2 presents the semantics of Petri nets, i.e., the principles of the dynamic behavior of Petri nets. Afterwards, Section 2.3.3 tells basic properties that characterize dynamic aspects of Petri nets. Finally, Section 2.3.4 discusses the structural class of Petri nets, in particular free-choice nets.

2.3.1. Definition of a Petri Net

Petri nets originate from the doctoral thesis of Carl Adam Petri [104], where the author generalizes automata theory by concurrency. Since then, the research of Petri nets has led to a large body of formal results and applications. For a review of the history of Petri nets please refer to [95]. The classical Petri net is a directed bipartite graph with two types of nodes called places and transitions. The nodes of a Petri net are connected via directed arcs; arcs never connect two places or two transitions. In this section we present standard definitions of Petri nets.

Definition 2.7 (Petri net).
A *Petri net*, or a *net*, is a tuple $N = (P, T, F)$, with P and T as finite disjoint sets of *places* and *transitions*, and $F \subseteq (P \times T) \cup (T \times P)$ as the *flow* relation. ⌟

We use subscripts P_N, T_N, and F_N to denote the relation of the sets to the net N. Note that we omit subscripts where the context is clear. We identify F with its characteristic function on the set $(P \times T) \cup (T \times P)$. We refer to the set $P \cup T$ as *nodes* of the net. For a node $x \in P \cup T$, $\bullet x = \{y \in P \cup T \mid F(y, x) = 1\}$ is a *preset* and $x\bullet = \{y \in P \cup T \mid F(x, y) = 1\}$ is a *postset* of x. A place $p \in P$

is a *source* place if $\bullet p = \varnothing$, and it is a *sink* place if $p\bullet = \varnothing$. By $Min(N)$ we denote the set of all source places of N, i.e., $Min(N)$, $N = (P, T, F)$, is the set $\{p \in P \mid \bullet p = \varnothing\}$. A node $x \in P \cup T$ is an *input* (*output*) node of a node $y \in P \cup T$ iff $x \in \bullet y$ ($x \in y\bullet$). For $X \subseteq P \cup T$, $\bullet X = \bigcup_{x \in X} \bullet x$ and $X\bullet = \bigcup_{x \in X} x\bullet$. For a node $x \in P \cup T$, $in(x) = \{(n, x) \in F \mid n \in \bullet x\}$ is the set of its *incoming* arcs and $out(x) = \{(x, n) \in F \mid n \in x\bullet\}$ is the set of its *outgoing* arcs. We denote by F^+ the transitive closure, and by F^* the reflexive and transitive closure of F.

Petri nets can be treated as directed graphs. Given a net $N = (P, T, F)$, the ordered pair $(P \cup T, F)$ defines a directed graph. The graph defines the structure of the net. In the graphical notation, it is widely accepted that places are represented by circles, transitions by rectangles, and flow relation by directed edges. Figure 2.4 shows a net composed of eight places $p_1, \ldots, p_8$ and eight transitions $t_1, \ldots, t_8$. We expect that all the nets that we shall work with are *T-restricted*, i.e., $\forall\ t \in T : \bullet t \neq \varnothing \neq t\bullet$. This often allows to avoid minor technical difficulties. If a net is not T-restricted, we assume its natural completion, i.e., the net gets modified so that a transition without an input (output) place gets a single input (output) place. The net in Figure 2.4 is T-restricted.

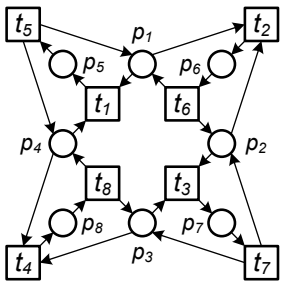


Figure 2.4. A net

It is often useful to distinguish between observable and silent transitions of a net. To cope with this phenomenon, the notion of a net is extended.

Definition 2.8 (Labeled net).
A *labeled* net is a tuple $N = (P, T, F, \mathcal{T}, \lambda)$, where (P, T, F) is a net, $\mathcal{T}$ is a finite set of *labels* such that $\tau \in \mathcal{T}$, and labeling $\lambda : T \to \mathcal{T}$ assigns to each transition a label. If $\lambda(t) \neq \tau$, $t \in T$, then t is *observable* in N; otherwise t is *silent*. Labeling λ is *distinctive* if it is injective on a subset of observable transitions. ⌟

Observable transitions are designed to represent actions of the distributed system that are visible to the outside world, while silent transitions are the internal transitions of the system. Figure 2.5 shows a labeled Petri net $N = (P, T, F, \mathcal{T}, \lambda)$, where $P = \{p_1, p_2, p_3\}$, $T = \{t_1, t_2, t_3, t_4\}$, $F = \{(p_1, t_1), (p_1, t_2), (t_1, p_2), (t_2, p_2), (p_2, t_3), (p_2, t_4), (t_3, p_3), (t_4, p_3)\}$, $\mathcal{T} = \{a, b, \tau\}$. The function λ is such that $\lambda(t_1) = a$, $\lambda(t_4) = b$, and $\lambda(t_2) = \lambda(t_3) = \tau$. In Figure 2.5, the silent transitions t_2 and t_3 are drawn as empty rectangles. Labels are positioned next to the corresponding observable transitions.

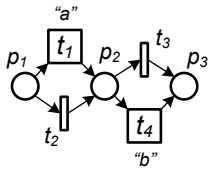


Figure 2.5. A labeled net

In our subsequent discussions we shall refer to parts of nets. Therefore, we formally define the notions of a subnet and path of a net. Let $N' = (P', T', F')$ and $N = (P, T, F)$ be two nets, such that $P' \subseteq P$, $T' \subseteq T$. N' is a *subnet* of N, denoted by $N' \subseteq N$, iff $F' = F \cap ((P' \times T') \cup (T' \times P'))$. Subnet N' is said to be *induced* by nodes $P' \cup T'$. N' is a *partial subnet* of N, denoted by $N' \leq N$, iff $F' \subseteq F \cap ((P' \times T') \cup (T' \times P'))$. A *path* of a net $N = (P, T, F)$ is a non-empty

sequence $x_1, \ldots, x_k$ of nodes, $x_i \in P \cup T$, $1 \leq i \leq k$ and $k > 1$, denoted by $\pi_N(x_1, x_k)$, which satisfies $(x_1, x_2), \ldots, (x_{k-1}, x_k) \in F$. We write $x_i \in \pi_N$ if x_i is on the path π_N. A *subpath* π'_N of a path π_N is a subsequence of π_N, which is itself a path. Sometimes we shall identify (sub)paths with the sets of their nodes.

A Petri net is a static model of a distributed system. In the next section, we explain how Petri nets capture the dynamic aspects of distributed systems.

2.3.2. Semantics of Nets

The dynamics of a distributed system are determined by its states and the principles of state transitions. Petri nets have a precise execution semantics which is defined in terms of a token game. To describe the state of a Petri net, its places may contain tokens. The distribution of tokens over the places of the net is referred to as the *marking* of the net. Therefore, the state of the distributed system is uniquely determined by the marking of the corresponding Petri net.

Definition 2.9 (Marking).
Let $N = (P, T, F)$ be a net. A *marking*, or a *state*, of N is a function $M : P \to \mathbb{N}_0$ which assigns to each place a natural number of *tokens* (including zero). ⌟

We denote by $[p]$, $p \in P$, the marking M in which place p contains just one token and all other places contain no tokens, i.e., $M(p) = 1$ and $\forall\, p' \in P \setminus \{p\} : M(p') = 0$. We identify M with the multi-set containing $M(p)$ copies of p for every $p \in P$.

Definition 2.10 (Net system).
A *net system*, or a *system*, is a pair $S = (N, M)$, where N is a net and M is a marking of N. ⌟

We denote by M_0 the *initial* marking of N. We use $\hat{M}$ to denote the *natural* marking of N. The natural marking of a net puts one token at every place without incoming arcs and no tokens elsewhere. In the following, when we refer to a net as a system, we assume the net in its natural state.

The marking of a net implies a subset of enabled transitions of the net. Intuitively, enabled transitions can be interpreted as the *actions* of the distributed system that are ready to be executed.

Definition 2.11 (Transition enabling).
Let $S = (N, M)$, $N = (P, T, F)$, be a net system. A transition $t \in T$ is *enabled* in S, denoted by $(N, M)[t\rangle$, iff every place from the preset of t contains at least one token, i.e., $\forall\, p \in {\bullet t} : M(p) \geq 1$. ⌟

When a transition of a net is enabled, the net can *fire* this transition. If the transition fires, then one token is removed from every input place and one token is added to every output place of this transition.

Definition 2.12 (Firing rule).
Let $S = (N, M)$, $N = (P, T, F)$, be a net system. If a transition $t \in T$ is enabled in S then it can *fire* which leads to a new marking M'. The new marking M' is defined by $M'(p) = M(p) - F(p, t) + F(t, p)$, for each place $p \in P$. The firing is denoted by $(N, M)[t\rangle(N, M')$. ⌟

The firing of a transition consumes no time and brings the system from one state to the other state, which constitutes a state transition. Firing is nondeterministic, i.e., if multiple transitions are enabled at the same time, then any one of them may fire. Because multiple tokens may be present in the net, Petri nets are well suited for modeling the concurrent behavior of distributed systems.

The firing rule determines the set of *reachable markings* of a system. A marking M is reachable from the initial marking M_0 of a system (N, M_0), if and only if there exists a sequence of enabled transitions whose firings lead from M_0 to M.

Definition 2.13 (Reachable marking).
Let $S = (N, M_0)$, $N = (P, T, F)$, be a net system. A sequence of transitions $\sigma = t_1, \ldots, t_n$, $n \in \mathbb{N}_0$, is a *firing sequence* in S iff there exists a sequence of firings $(N, M_0)[t_1\rangle(N, M_1), \ldots, (N, M_{n-1})[t_n\rangle(N, M_n)$ which leads from marking M_0 to marking M_n via a (possibly empty) sequence of intermediate markings $M_1, \ldots, M_{n-1}$. For any two markings M and M', M' is *reachable* from M, denoted by $M' \in [N, M\rangle$, iff there exists a firing sequence σ leading from M to M'. ⌟

The set $[N, M_0\rangle$ contains all reachable markings of the system and thus defines its state space. Next, we exemplify the presented concepts.

The Dining Philosophers Problem – An Illustrative Example

In computer science, the dining philosophers problem is an illustrative example of a multi-process synchronization problem. It was originally described by Edsger Dijkstra as a problem where five computers concurrently compete for access to five shared tape drives. Later, Tony Hoare reformulated the problem into the dining philosophers problem [52].

The dining philosophers problem is formulated as a number of philosophers sitting together at a round table and each doing one of two things, either eating or thinking. There is a fork on the table between each pair of philosophers who are sitting next to each other. In order to start eating, a philosopher needs to pick up two forks. The philosopher can only use the fork to his left and the fork to his right. Once the philosopher is finished with eating, the forks are placed back on the table. If a philosopher is not eating, (s)he is thinking, and vice versa. All the philosophers proceed with alternating eating and thinking phases independently from each other.

In the following, we show how the dining philosophers problem, in fact a slightly simplified one, can be formalized as a net system. The classical problem is proposed for $n = 5$ philosophers. The net in Figure 2.4 defines the static structure of our solution to the problem for $n = 4$. In order to define the dynamic aspects of the problem, we enhance the net with the initial marking; the resulting system is shown in Figure 2.6(a). Each token of the marking is visualized as a black dot inside of the corresponding place. Thus, the initial marking of the system in Figure 2.6(a) is $M_0 = \{p_1, p_2, p_3, p_4\}$.

For each place p_1, p_2, p_3, and p_4, the presence of a token in the place models the presence of the fork on the table. Thus, in the initial state, four forks are on the table. In this state there are four enabled transitions: t_1, t_2, t_3, and t_4; all

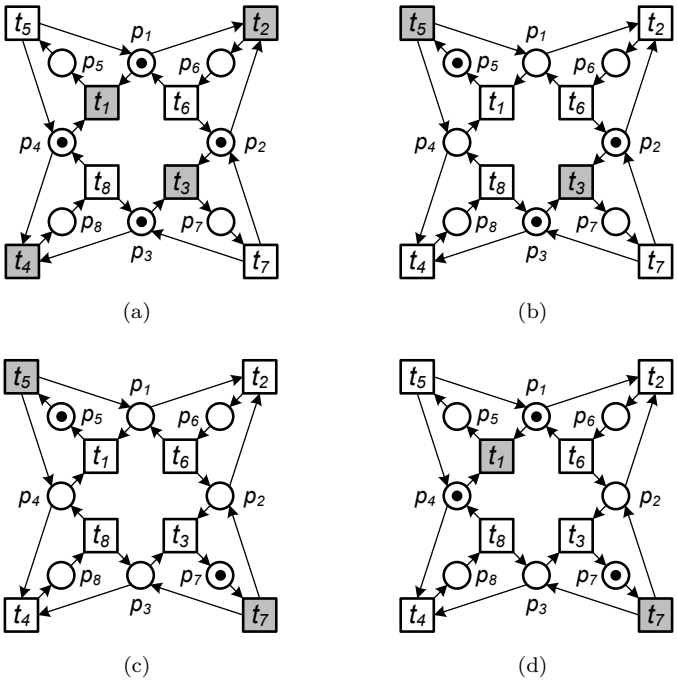


Figure 2.6. The simplified version of the dining philosophers problem ($n = 4$) formalized as a net system, given at the initial and the three different reachable markings

marked with grey background in Figure 2.6(a). Each of these transitions may fire; the firing models the philosopher picking up two forks next to him. Assume that t_1 fires nondeterministically. The firing brings the system to the new marking $\{p_2, p_3, p_5\}$, which is shown in Figure 2.6(b). For each place p_5, p_6, p_7, and p_8, the presence of a token in the place models the philosopher in the eating phase, while the absence represents the philosopher in the thinking phase. In the system in Figure 2.6(b) transitions t_3 and t_5 are enabled (highlighted with grey background). The firing of transition t_5, similar to the firing of transitions t_6, t_7, and t_8, models the shift from the eating to the thinking phase. However, transition t_3 fires, which results in the system shown in Figure 2.6(c). The resulting system describes two philosophers in the eating and two in the thinking phase. In the next step, transition t_5 fires and brings the system to the marking depicted in Figure 2.6(d).

The proposed solution can proceed infinitely long by following the execution semantics of Petri nets. In our solution, each philosopher picks up and puts back two forks simultaneously. Additionally, the phase, either the eating or the thinking, of each philosopher can be monitored by tracking the availability of a token at a single place. The classical dining philosophers problem assumes that philosophers operate with forks independently. Also, one can think of a solution in which two phases in the life of a philosopher are modeled as a single token alternating

between two places of a net. The reader is kindly advised to design different solutions for different settings and numbers of philosophers as an exercise; please label transitions in your net systems.

2.3.3. Basic Properties of Net Systems

Net systems are the models of real world or designed distributed systems. The primary use of models is the study of their properties. In this section, we discuss the basic properties of net systems which characterize their behavior.

The first property we describe is *boundedness* of a system.

Definition 2.14 (Boundedness).
A net system (N, M_0) is *bounded* iff the set of all its reachable markings $[N, M_0\rangle$ is finite; otherwise the system is *unbounded*. ⌟

Alternatively, a net system is bounded if there exists a number k, such that no reachable marking puts more than k tokens in any place. In an unbounded net system, the number of tokens in some places can grow infinitely large.

The *safeness* property of a net system restricts boundedness by requiring that there is never more than one token in the same place of the net.

Definition 2.15 (Safeness).
A net system (N, M_0) is *safe* iff for every reachable marking $M \in [N, M_0\rangle$ and for every place $p \in P$ the amount of tokens in p is not more than one, i.e., $\forall\ M \in [N, M_0\rangle\ \forall\ p \in P : M(p) \leq 1$; otherwise the system is *unsafe*. ⌟

The net system in Figure 2.6 is safe and, thus, bounded. The state space of the system consists of seven states. Every reachable marking of the system has at most one token in every place.

Another property of a net system is *liveness*. A net system is live if every transition can always fire again.

Definition 2.16 (Liveness).
A net system (N, M_0) is *live* iff for every marking M reachable from M_0 and for every transition $t \in T$, there exists a marking M' reachable from M which enables t, i.e., $\forall\ M \in [N, M_0\rangle\ \forall\ t \in T\ \exists\ M' \in [N, M\rangle : (N, M')[t\rangle$; otherwise the system is *not* live. ⌟

The net system in Figure 2.6 is live. It can always reach a marking which enables any transition from any reachable marking.

2.3.4. Structural Classes of Nets

Petri nets have a great expressive power. They can be used to model a large variety of distributed systems. However, the generality of the modeling language often inflicts high complexity on the analysis techniques for checking properties of designed systems. Hence, often, theoretic investigations are carried out for structural subclasses of *general* nets. In this section, we discuss several widely

used structural classes of nets and their relations. For details on the content that follows, the reader is kindly forwarded to [12, 10, 22].

We start our discussion with two basic classes of nets: S-nets and T-nets.

Definition 2.17 (S-net, T-net).
Let $N = (P, T, F)$ be a net.

- N is an *S-net* iff every transition $t \in T$ has exactly one input place and one output place, i.e., $\forall\ t \in T : |\bullet t| = 1 = |t \bullet|$.
- N is a *T-net* iff every place $p \in P$ has exactly one input transition and one output transition, i.e., $\forall\ p \in P : |\bullet p| = 1 = |p \bullet|$. ⌟

A system (N, M_0) is an *S-system* (a *T-system*) if N is an S-net (a T-net). The fundamental property of an S-system is that all its reachable markings contain the same number of tokens. Therefore, if the initial marking of an S-system contains only one token, the system can be interpreted as a *state machine*. T-systems allow concurrency and synchronization, but no conflicts. T-systems are also known in literature as *marked graphs*.

Free-choice nets are a common generalization of S-nets and T-nets. In a free-choice net, two places that share an output transition may not have any other output transitions and two transitions that share an input place may not have any other input places.

Definition 2.18 (Free-choice net).
A net $N = (P, T, F)$ is *free-choice* iff $\forall p \in P,\ |p \bullet| > 1 : \bullet(p\bullet) = \{p\}$. ⌟

Again, a system (N, M_0) is free-choice if N is free-choice. Apparently, every S-net and every T-net is free-choice and, thus, the class of free-choice nets is indeed the generalization of both. The free-choice property guarantees that if two transitions share an input place then every reachable marking of the system enables either both of these transitions or none of them. Therefore, the choice of firing a transition from the set of enabled transitions is not influenced by the rest of the system. Hence the name – free-choice.

The very same effect on choices of transition firings can be achieved with less restrictions on the structure of nets. In *extended* free-choice nets, if there is a flow from a place p to a transition t, then there must be a flow from any input place of t to any output transition of p.

Definition 2.19 (Extended free-choice net).
A net $N = (P, T, F)$ is *extended* free-choice, iff $\forall t_1, t_2 \in T : \bullet t_1 \cap \bullet t_2 \neq \varnothing \Rightarrow \bullet t_1 = \bullet t_2$. ⌟

It is known that every extended free-choice system can be "simulated" by a free-choice one, see [12, 10]. The simple construction to convert an extended free-choice net into an "equivalent" free-choice net is shown in Figure 2.7. The transformation introduces a fresh place and a fresh transition to the resulting net. Under equivalence we understand that any behavior of the resulting free-choice system may be interpreted as some behavior of the original extended free-choice system if fresh elements are ignored.

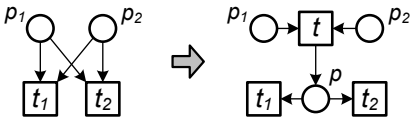


Figure 2.7. A transformation which reduces extended free-choice nets to free-choice nets

A common practice in literature is to refer to extended free-choice nets simply as free-choice, e.g., see [22]. In the following, we obey historical correctness and only refer to nets as free-choice if they comply with Definition 2.18. Figure 2.8(a) shows an extended free-choice net. The net in Figure 2.8(b) is the free-choice net obtained from Figure 2.8(a) by applying the transformation from Figure 2.7.

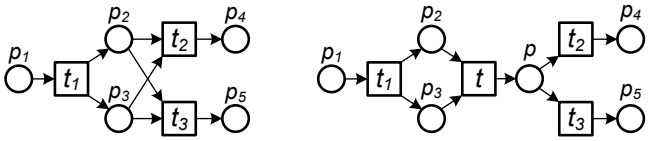


Figure 2.8. (a) An extended free-choice net and (b) the equivalent free-choice net

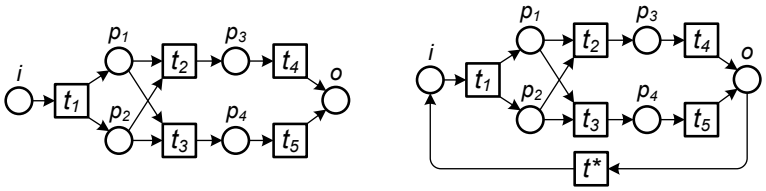
Observe that the net in Figure 2.5 is an S-net and the net in Figure 2.4 is neither extended free-choice, nor free-choice. In the sequel, we will mostly work with free-choice nets as per Definition 2.18. Note that all the results that will be discussed for free-choice nets are directly applicable to net classes that are reducible to free-choice nets, e.g., extended free-choice nets or behaviorally free-choice nets [10].

2.4. Workflow Nets

Workflow (WF-)nets are the structural subclass of Petri nets specifically designed to represent workflow procedures [129, 130]. A WF-net is a net with two special places: one place is used to mark the beginning and the other one is used to mark the termination of a workflow procedure. In Section 2.4.1, we give the definition of WF-nets, whereas in Section 2.4.2, we discuss the soundness property – a behavioral correctness property every proper workflow procedure should satisfy.

2.4.1. Definition of a Workflow Net

A net system starts when the first transition fires and terminates once no transition is enabled. Workflow systems are the net systems which explicitly model the state in which no transition has yet fired and the state in which no transition should be enabled. The described design is achieved by imposing structural restrictions on the nets of workflow systems.



(a) (b)

Figure 2.9. (a) A WF-net and (b) its short-circuit net

Definition 2.20 (WF-net, Short-circuit net, WF-system).
A net $N = (P, T, F)$ is a *workflow net*, or a *WF-net*, iff N has a dedicated source place $i \in P$, N has a dedicated sink place $o \in P$, and the *short-circuit* net $N' = (P, T \cup \{t^\star\}, F \cup \{(o, t^\star), (t^\star, i)\})$, $t^\star \notin T$, of N is strongly connected. A *WF-system* is a pair (N, M_i), where $M_i = [i]$. ⌟

A workflow system gets initialized once there is a token in the source place and terminates once there is a token in the sink place. In the following, we always assume that the source place and the sink place are denoted by $i \in P$ and, respectively, $o \in P$. Similar to M_i, we shall denote marking $[o]$ by M_o.

The net in Figure 2.5 is a workflow net with source p_1 and sink p_3, while the net in Figure 2.8(a) is *not* a WF-net – it is impossible to identify a single dedicated sink place. In Figure 2.9(a), we complete the net in Figure 2.8(a) to a WF-net. Finally, Figure 2.9(b) shows its short-circuit net with a fresh transition $t^\star$.

2.4.2. Soundness

The requirements stated in Definition 2.20 relate to the structure of nets and can be checked by employing standard graph techniques. In this section, we discuss the soundness property, which was introduced in [129], that relates to the dynamics of WF-systems. A sound WF-system must eventually terminate, and once it terminates, there must be a single token in its sink place and no token elsewhere. Additionally, in a sound system it should be possible to fire any transition by following some firing sequence through the net. Let M and M' be two markings of a net (P, T, F); $M \geq M'$ iff $\forall\ p \in P : M(p) \geq M'(p)$. The soundness property is then formalized as follows.

Definition 2.21 (Soundness).
A workflow system (N, M_i), $N = (P, T, F)$, is *sound* iff :

- For every marking M reachable from M_i, there exists a firing sequence leading from marking M to marking M_o, i.e., $\forall M \in [N, M_i\rangle : M_o \in [N, M\rangle$.
- Marking M_o is the only marking reachable from M_i with at least one token in place o, i.e., $\forall M \in [N, M_i\rangle, M \geq M_o : M = M_o$.
- Every transition can be enabled, i.e., $\forall\ t \in T\ \exists\ M \in [N, M_i\rangle : (N, M)[t\rangle$. ⌟

If a WF-system is not sound, we refer to it as *unsound*. The soundness property is summarized in three requirements: (i) It is always possible to reach marking

M_o starting from the initial marking M_i. (ii) Once a token appears at the sink place, all other places of the net must be empty. (iii) Every transition can be enabled in some marking that is reachable from the initial marking. Considering the nondeterministic nature of the firing rule, the last requirement implies the fact that every transition can fire.

The problem of deciding whether a given WF-net is sound corresponds to the problem of checking standard properties of Petri nets. In [129], Wil van der Aalst shows that the soundness of a workflow net corresponds to the liveness and boundedness of its short-circuit net. This result is proposed in the next theorem.

Theorem 2.22 ([129], Theorem 11).
A WF-system (N, M_i) is sound, iff (N', M_i) is live and bounded, where N' is the short-circuit net of N. ⋆

Theorem 2.22 allows one to organize soundness verification of a WF-system using standard techniques for checking properties of net systems. A net system composed from the net in Figure 2.9(b) and marking M_i is live and bounded and, hence, the WF-net in Figure 2.9(a) is sound.

2.5. Process Models

In the Introduction (Chapter 1), we discussed behavioral models rather informally and stated that we shall instantiate the structuring problem with one concrete notion of a behavioral model, which we refer to as a *process model*. In this section, we formally define process models. We keep the formalism concise and sufficient for the later discussions. Section 2.5.1 gives the definition of a process model, while Section 2.5.2 discusses execution semantics of process models.

2.5.1. Definition of a Process Model

The notion of a process model is the outcome of our attempts to define a greatest common divisor for common languages that are used to describe behavioral models. Additionally, the notion of a process model is the minimal yet sufficient concept for the discussion of the structuring problem, which we shall commence in Part III. We consider process models as captured in the following definition.

Definition 2.23 (Process model).
A *process model* is a tuple $PM = (A, G, C, type, \mathcal{A}, \mu)$, with A and G as finite disjoint sets of *tasks* and *gateways*, respectively, and $C \subseteq (A \cup G) \times (A \cup G)$ as the *control flow* relation. $type : G \to \{xor, and\}$ assigns a type to each gateway. $\mathcal{A}$ is a finite set of *names*, such that $\tau \in \mathcal{A}$, and naming $\mu : A \to \mathcal{A}$ assigns to each task a name. If $\mu(a) \neq \tau$, $a \in A$, then a is *observable* in PM; otherwise a is *silent*. ⌟

We use subscripts, e.g., A_{PM} or G_{PM}, to denote relation of the sets to the process model PM and omit subscripts where the context is clear. We refer to the set $A \cup G$ as *nodes* of the process model. A node $x \in A \cup G$ is a *source* (*sink*) iff $\bullet x = \varnothing$

($x\bullet = \varnothing$), where $\bullet x$ ($x\bullet$) stands for the set of direct predecessors (direct successors) of x in the directed graph $(A \cup G, C)$.

In the following, we assume that process models satisfy certain structural requirements: We expect that source and sink nodes of a process model are tasks; otherwise we assume its natural completion, i.e., the process model gets modified so that a source (sink) gateway gets a fresh direct predecessor (direct successor) which is a silent task. We expect that every node of PM lies on a path from a source to a sink. Each task $a \in A$ has at most one incoming and at most one outgoing arc, i.e., $|\bullet a| \leq 1 \wedge |a\bullet| \leq 1$. Each gateway has at least three incident control flow arcs.

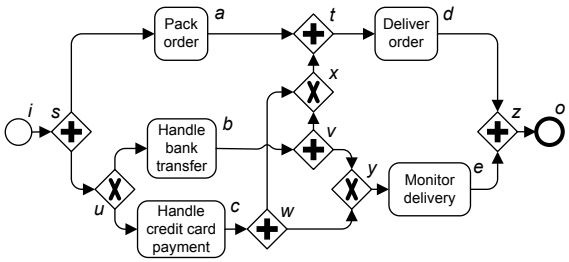


Figure 2.10. A process model

Figure 2.10 shows a process model which describes a simple "order delivery and payment" process. We adapt the notation similar to BPMN for visualization of process models. The process model in Figure 2.10 contains seven tasks, $A = \{i, a, b, c, d, e, o\}$. Note that i and o are silent tasks, i.e., $\mu(i) = \tau = \mu(o)$. We visualize silent tasks as start, intermediate, or end events in BPMN [2]. Source silent tasks are drawn as BPMN start events, sink silent tasks are drawn as BPMN end events, and all other silent tasks are drawn as BPMN intermediate events. Silent tasks serve a technical purpose, e.g., representation of source and sink tasks of a process model. An observable task is drawn as a rectangle that has rounded corners with its name inside. Gateways are visualized as diamonds: Gateways of type *xor*, or *exclusive* gateways, use a marker which is shaped like an "×" inside the diamond shape. Gateways of type *and*, or *parallel* gateways, use a marker which is shaped like a "+" inside the diamond shape. A gateway $g \in G$ is a *split* iff $|\bullet g| = 1 \wedge |g\bullet| > 1$. A gateway $g \in G$ is a *join* iff $|\bullet g| > 1 \wedge |g\bullet| = 1$. The set $\{s, t, u, v, w, x, y, z\}$ is the set of gateways of the process model in Figure 2.10. Gateways u, x, and y are exclusive, while gateways s, t, v, w, and z are parallel. Gateways s, u, v, and w are splits, while gateways t, x, y, and z are joins. Finally, control flow arcs are drawn as directed edges between tasks and gateways.

2.5.2. Semantics of Process Models

Process models as per Definition 2.23 are static directed graphs with typed nodes. In this section, we present the execution semantics of process models. The execution semantics describes the dynamics of process models.

A task in a process model is an atomic piece of work. The execution of a task requires time. Tasks are usually performed by human process participants and/or automated systems. Gateways are used to control process execution. Unlike tasks, gateways do not represent work to be done and have zero effect on execution time. Exclusive gateways are used to create alternative threads of control when executing process models. An exclusive gateway passes the thread of control to one of its outgoing control flow arcs every time the execution of the process model reaches the gateway along one of its incoming control flow arcs. The selection of an outgoing arc is usually associated with a *decision*. Parallel gateways are used to synchronize or to create parallel threads of control. A parallel gateway passes one thread of control to each of its outgoing control flow arcs every time the execution of the process model reaches the gateway along all its incoming control flow arcs. Control flow arcs define the ordering constraints between tasks and gateways of the process model.

The formal definition of the execution semantics of process models is carried out by means of a mapping to labeled Petri nets. As an outcome, the execution semantics of the resulting Petri net (in terms of its token game) defines the execution semantics of the process model.

Definition 2.24 (Net of a process model).
Let $PM = (A, G, C, type, \mathcal{A}, \mu)$ be a process model and let $G^{\times}$ and G^{+} denote exclusive and parallel gateways, respectively, i.e., $G^{\times} = \{g \in G \mid type(g) = xor\}$ and $G^{+} = \{g \in G \mid type(g) = and\}$. Let $I \subseteq A$ and $O \subseteq A$ be sources and sinks of PM, respectively, i.e., $I = \{x \in A \mid \bullet x = \varnothing\}$ and $O = \{x \in A \mid x\bullet = \varnothing\}$. The labeled net $N = (P, T, F, \mathcal{T}, \lambda)$ that *corresponds* to PM is defined by:

- $P = \{p_x \mid x \in G^{\times}\} \cup \{p_{x,y} \mid (x,y) \in C \wedge y \in A \cup G^{+}\} \cup \{p_x \mid x \in I \cup O\}$.
- $T = \{t_x \mid x \in A \cup G^{+}\} \cup \{t_{x,y} \mid (x,y) \in C \wedge x \in G^{\times}\}$.
- $F = \{(t_x, p_y) \mid (x,y) \in C \wedge x \in A \cup G^{+} \wedge y \in G^{\times}\} \cup \{(t_x, p_{x,y}) \mid (x,y) \in C \wedge x, y \in A \cup G^{+}\} \cup \{(t_{x,y}, p_y) \mid (x,y) \in C \wedge x, y \in G^{\times}\} \cup \{(t_{x,y}, p_{x,y}) \mid (x,y) \in C \wedge x \in G^{\times} \wedge y \in A \cup G^{+}\} \cup \{(p_x, t_{x,y}) \mid (x,y) \in C \wedge x \in G^{\times}\} \cup \{(p_{x,y}, t_y) \mid (x,y) \in C \wedge y \in A \cup G^{+}\} \cup \{(p_x, t_x) \mid x \in I\} \cup \{(t_x, p_x) \mid x \in O\}$.
- $\mathcal{T} = \mathcal{A} \cup \{\tau\}$. $\lambda(t_x) = \mu(x), t_x \in T, x \in A$; otherwise $\lambda(t) = \tau, t \in T$. ⌟

Definition 2.24 is similar to Definition 2.18 in [68]. However, Definition 2.24 tends to generate smaller nets in terms of the number of nodes. The definition states that a task is mapped to a Petri net transition with a single input and a single output flow arc. A parallel gateway maps to a transition with multiple incoming flow arcs and/or multiple outgoing flow arcs. An exclusive gateway maps to a place with multiple incoming flow arcs and/or multiple outgoing flow arcs. The places that correspond to *xor* gateways with multiple outgoing control flow arcs are immediately followed by silent transitions which represent decisions. Sources and sinks of the process model are mapped to places. The proposed mapping formalizes the execution semantics of process models. It is easy to see that the mapping always results in free-choice nets.

Figure 2.11 shows the net which corresponds to the process model in Figure 2.10. Note that the resulting net is not minimal in terms of the number of nodes. We propose the reader an exercise. Please define a mapping that defines the execution

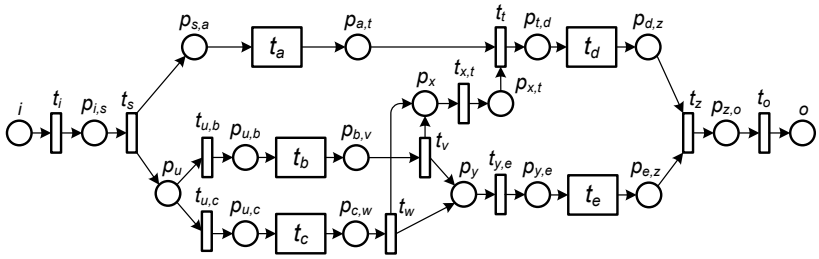


Figure 2.11. A WF-net that corresponds to the process model in Figure 2.10

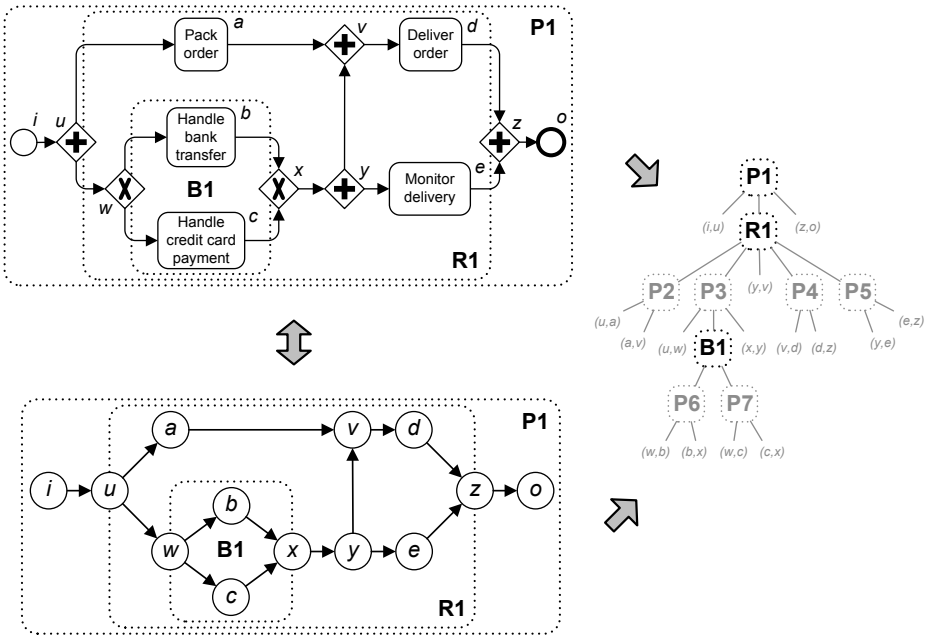
semantics of process models (as proposed above) and which results in free-choice nets smaller than those produced by Definition 2.24. As we require any such mapping, in the following we shall work with nets generated by Definition 2.24.

We refer to a process model as *single-source and single-sink* if it has only one source task and only one sink task. Observe that the net which corresponds to a single-source and single-sink process model is always a workflow net, e.g., the net in Figure 2.11. We propose to deduce the properties of process models from the properties of corresponding nets. For instance, a single-source and single-sink process model is *sound* if the corresponding WF-net is sound. Note that a sound free-choice WF-system is guaranteed to be safe [131]. Hence, a sound single-source and single-sink process model is also safe.

Part II.

Parsing and Abstraction

3. Parsing



Behavioral models are often formalized as directed graphs of a special kind, viz. *workflow graphs*. Workflow graphs can be parsed into subgraphs with a single entry and single exit. The result of the parsing procedure is a *parse tree*, which is the compositional containment hierarchy of the subgraphs. Jussi Vanhatalo, Hagen Völzer, and Jana Koehler [145, 146] proposed a workflow graph parsing technique called the *Refined Process Structure Tree* (RPST). In this chapter, we propose an alternative way to compute the RPST that is simpler than the one developed originally. Section 3.1 presents the notion of the RPST. Afterwards, Section 3.2 discusses connectivity related notions of graphs. These notions are then used in Section 3.3 to propose a new technique for the RPST computation. Section 3.4 shows how the RPST computation can be generalized to become applicable to any workflow graph. Finally, Section 3.5 discusses related work and states a conclusion.

The materials reported in this chapter are published in [111, 113].

3.1. The Refined Process Structure Tree

As stated in the Introduction (Chapter 1), in this thesis, we study behavioral models that are formalized as directed graphs; BPMN models, EPCs, activity diagrams, Petri nets (Section 2.3), process models (Section 2.5), are examples of languages that suggest to employ directed graphs when describing behaviors. Often, graphs used to capture behavioral models are expected to comply with basic structural correctness requirements. In this chapter, we refer to such graphs as *workflow graphs*. The Refined Process Structure Tree (RPST) is a technique for workflow graph *parsing*, i.e., for discovering the structure of a workflow graph, with desirable properties and various applications. In Section 3.1.1 we give an introduction to the parsing problem. Afterwards, Section 3.1.2 and Section 3.1.3 formally define the notions of a workflow graph and the RPST, respectively.

3.1.1. About Workflow Graph Parsing

Companies widely use behavioral models for documenting their operational procedures. Analysts develop behavioral models by decomposing business scenarios into tasks and defining their logical and temporal dependencies. The models are then utilized for communicating, analyzing, optimizing, and supporting execution of individual business cases within or across companies. Various modeling notations have been proposed. Many of them, for example the Business Process Modeling Notation (BPMN), Event-driven Process Chains (EPC), and UML activity diagrams, are based on *workflow graphs*, which are directed graphs with nodes representing activities or control decisions, and edges specifying temporal dependencies.

A workflow graph can be parsed into a hierarchy of subgraphs with a single entry and single exit. Such a subgraph is a logically independent subworkflow, or subprocess, of the behavioral model. The result of the parsing procedure is a *parse tree*, which is the containment hierarchy of the subgraphs. The parse tree has various applications, e.g., translation between process languages [146, 45, 109], control-flow and data-flow analysis [41, 42, 63, 62, 147], process comparison and merging [74], process abstraction [111], process comprehension [148], model layout [6], pattern application in process modeling [47], etc.

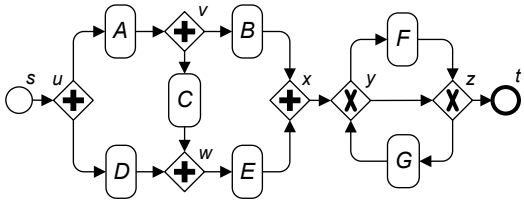
Jussi Vanhatalo, Hagen Völzer, and Jana Koehler [145, 146] proposed a workflow graph parsing technique, called the *Refined Process Structure Tree* (RPST), that has a number of desirable properties: The resulting parse tree is unique and *modular*, where *modular* means that a local change in the workflow graph only results in a local change of the parse tree. Furthermore, it is finer grained than any known alternative approach and it can be computed in linear time. The linear time computation is based on the idea by Robert Endre Tarjan and Jacobo Valdes [124] to compute a parse tree based on the *triconnected components* of a biconnected graph.

The original RPST algorithm [145] contains, besides the computation of the triconnected components, a post-processing step that is fairly complex. In the following, we shall show that the computation can be considerably simplified by introducing a pre-processing step that splits every node of the workflow graph with

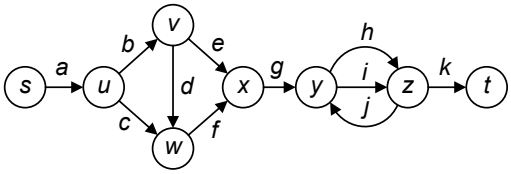
more than one incoming and more than one outgoing edge into two nodes. We prove that for the resulting graph, the RPST and the triconnected components coincide. Furthermore, we prove that the RPST of the original graph can then be obtained by a simple post-processing step. This new approach reduces the implementation effort considerably, requiring only little more than the computation of the triconnected components, of which an implementation is publicly available [48].

3.1.2. Workflow Graphs

We have already mentioned BPMN, EPC, and UML activity diagram modeling languages, and have formally discussed execution semantics of Petri nets (Section 2.3) and process models (Section 2.5). In a simplistic setting, models captured in these languages can be addressed as directed graphs with different types of nodes. Such simplicity is, however, sufficient for many applications. Usually, node types, as well as the concrete execution semantics of a model, are irrelevant for parsing purposes. Also, they are irrelevant for the discussion of the RPST. The only property of importance when parsing a model is its topology. To simplify the presentation and to stress the generality of the results, the following discussions are carried out for directed multi-graphs, which we call *workflow graphs*. Workflow graphs are not arbitrary graphs, they are derived from behavioral models and, thus, are expected to follow certain structural requirements.



(a)



(b)

Figure 3.1. (a) A process model and (b) its corresponding TTG (simplified)

Similar as in the case of Petri nets and process models, a node v of a directed (multi-)graph is a *source* (*sink*), iff it has no direct predecessors (direct successors). We distinguish two structural classes of directed graphs, see the next definition.

Definition 3.1 (Multi-terminal graph, Two-terminal graph).
A *multi-terminal graph* (*MTG*) is a directed multi-graph G that has at least one source and at least one sink, such that each node lies on a path from some

source to some sink; G is a *two-terminal graph* (TTG) if it has exactly one source and exactly one sink. ⌟

We refer to a directed multi-graph as a *workflow graph*, if it an MTG. Figure 3.1(a) shows a process model and Figure 3.1(b) presents the corresponding TTG. Note that the task nodes are skipped in the TTG for the sake of simplicity. Also, in the following, we assume for simplicity of the presentation that every MTG has at least two nodes and two edges.

Workflow graphs capture topologies of models and are, therefore, the minimal and sufficient concepts for the discussion of a parsing technique. Though we shall work with graphs, parsing can be performed on models which fulfill structural requirements of workflow graphs in a straightforward manner. Observe, for instance, that a workflow net or a single-source and single-sink process model is a TTG, while a process model, in general, is formalized as an MTG.

3.1.3. Definition of the Refined Process Structure Tree

After the introductory discussions, this section is devoted to the formal definition of the RPST. The RPST is a technique to parse workflow graphs into a collection of its subgraphs, each with a single entry and single exit. Therefore, we start the discussion with the formal definition of entries and exits of a subgraph.

Definition 3.2 (Interior, Boundary, Entry, and Exit nodes of a subgraph).
Let G be an MTG and $G_F = (V_F, F)$ be a connected subgraph of G that is formed by a set F of edges.

- A node $x \in V_F$ is *interior* with respect to G_F, iff it is connected only to nodes in V_F; otherwise v is a *boundary* node of G_F.
- A boundary node u of G_F is an *entry* of G_F, iff no incoming edge of u belongs to F or if all outgoing edges of u belong to F.
- A boundary node v of G_F is an *exit* of G_F, iff no outgoing edge of v belongs to F or if all incoming edges of v belong to F. ⌟

Figure 3.2 shows two subgraphs of the same TTG. Each subgraph is formed by two edges, which are inside of the corresponding area denoted by the dotted border, and has two boundary nodes y and z. Subgraph $S1$ is formed by edges $\{h, i\}$; y is the entry and z is the exit of $S1$. Subgraph $S2$ is formed by edges $\{i, j\}$; each of its boundary nodes cannot be classified neither as an entry, nor as an exit.

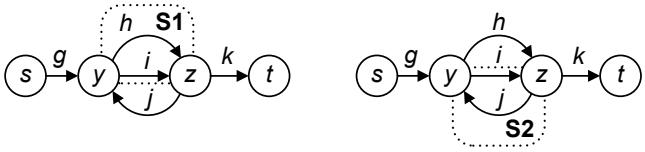
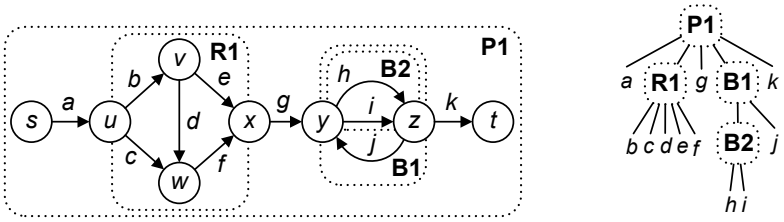


Figure 3.2. Subgraphs

A *fragment* of an MTG is a subgraph of the MTG with a special configuration of its boundary nodes.

Definition 3.3 (Fragment).
Let G be an MTG and $G_F = (V_F, F)$ be a connected subgraph of G that is formed by a set F of edges. F is a *fragment* of G, iff G_F has exactly two boundary nodes: one is an entry and the other one is an exit. ⌟

Subgraph $S1$ in Figure 3.2(a) is a fragment. The set $\{y, z\}$ containing the entry and the exit node is also called the *entry-exit pair* of the fragment. A fragment is *trivial* if it only contains a single edge. Note that every singleton edge forms a fragment. By definition, the source of a TTG is an entry to every fragment it belongs to and the sink of a TTG is an exit from every fragment it belongs to. Intuitively, control "enters" the TTG through the source and "exits" the TTG through the sink. Note also that we represent a fragment as a set of edges rather than as a subgraph (Section 2.1.1).



(a) (b)

Figure 3.3. (a) A TTG and its canonical fragments, (b) the RPST of (a)

We say that two fragments F, F' are *nested* if $F \subseteq F'$ or $F' \subseteq F$. They are *disjoint* if $F \cap F' = \emptyset$. If they are neither nested nor disjoint, we say that they *overlap*. The RPST of a workflow graph is a collection of its special fragments, viz. *canonical* fragments.

Definition 3.4 (Canonical Fragment).
Let G be an MTG. A fragment of G is *canonical*, if it does not overlap with any other fragment of G. ⌟

Finally, the RPST of a workflow graph is the set of all its canonical fragments. It is easy to see that due to the definition of a canonical fragment, the RPST is unique for a given graph. Moreover, canonical fragments of the RPST define all the fragments of the workflow graph as non-canonical fragments can be obtained as subsets of canonical ones, see [144] for more details.

Definition 3.5 (The Refined Process Structure Tree).
Let G be an MTG. The *Refined Process Structure Tree* (*RPST*) of G is the set of all canonical fragments of G. ⌟

It follows that any two canonical fragments are either nested or disjoint and, hence, they form a compositional containment hierarchy. This hierarchy can be shown as a tree, where the parent of a canonical fragment F is the smallest canonical

fragment that contains F. The root of the tree is the entire graph, the leaves are the trivial fragments.

Figure 3.3 exemplifies the RPST. Figure 3.3(a) shows the TTG from Figure 3.1(b) along with its canonical fragments, where every fragment is formed by edges enclosed in or intersecting an area denoted by the dotted border. For example, the canonical fragment $R1$ is formed by edges $\{b, c, d, e, f\}$, has interior nodes $\{v, w\}$ and boundary nodes $\{u, x\}$, with u being an entry and x an exit of the fragment. Figure 3.3(b) visualizes the RPST as a tree.

3.2. Connectivity, Decomposition, and Components

Prior that we start with the presentation of the main results on the simplified computation of the RPST, this section presents – at the sufficient level – all the required preliminary notions. Section 3.2.1 discusses the connectivity property of graphs. Section 3.2.2 shows how the connectivity property can be employed to perform graph decompositions. Finally, Section 3.2.3 is devoted to the presentation of the triconnected components of a graph.

3.2.1. Graph Connectivity

Connectivity is one of the basic concepts in graph theory. Given an undirected (multi-)graph G, two vertices u and v are *connected* in G, if graph G contains a path between u and v; otherwise the vertices are considered to be *disconnected*. A graph G is *connected*, if every pair of distinct vertices in G is connected; otherwise graph G is *disconnected*. Please note that though connectivity is defined for undirected (multi-)graphs, it can be used in the context of directed graphs in a straightforward manner, i.e., by ignoring edge directions.

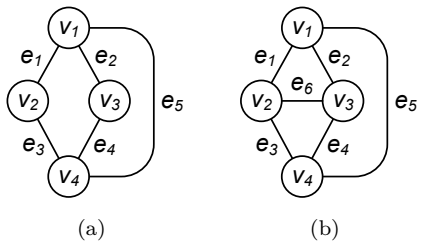


Figure 3.4. (a) A biconnected graph and (b) a complete graph

k-connectivity is the generalization of the connectivity property. A graph G is *k-connected* if there exists no set of $k-1$ elements, each a vertex or an edge, whose removal renders the graph disconnected, i.e., there is no path between some pair of elements in the graph. The set is called a *separating* $(k-1)$*-set* of G. Note that removal of a vertex implies removal of all its incident edges. Separating 1- and 2-sets of a graph that are composed solely of vertices are called *separation points* (or *cutvertices*) and *separation pairs*, respectively. 1-, 2-, and 3-connected graphs are referred to as *connected*, *biconnected*, and *triconnected*, respectively. Finally, the *connectivity* of a graph is the largest k for which the graph is k-connected. Note that a graph composed of a single vertex is accepted to be connected, while a complete graph that is composed of n vertices, $n \geq 2$, is $(n-1)$-connected.

Figure 3.4 shows two graphs. The graph in Figure 3.4(a) is clearly connected, but it is also biconnected. Observe that removal of any element of the graph, either a vertex or an edge, keeps the graph connected. However, the graph in Figure 3.4(a) is not triconnected. The removal of a separation pair $\{v_1, v_4\}$ renders the graph composed of two disconnected vertices v_2 and v_3. Hence, the largest k for which the graph is k-connected is two. The graph in Figure 3.4(a) can be modified to become "better" connected. The graph in Figure 3.4(b) is obtained from the graph in Figure 3.4(a) by adding a single edge that connects vertices v_2 and v_3. The modified graph is a complete graph composed of four vertices and, thus, it is triconnected. Note that removal of any pair of elements of the graph in Figure 3.4(b) renders a connected graph.

3.2.2. Connectivity-Based Decomposition of Graphs

A k-connected graph contains no separating $(k-1)$-sets, but can contain separating k-sets. After removing a separating set from a graph, the graph gets decomposed into disconnected subgraphs, or *components*. Subsequently, obtained subgraphs of higher connectivity can be decomposed by using larger separating sets. By gradually increasing the size of separating sets used to decompose a graph, one performs a stepwise connectivity-based decomposition of the graph. In Figure 3.5(b) and Figure 3.5(c), we exemplify two steps of the connectivity-based decomposition of the graph in Figure 3.5(a).

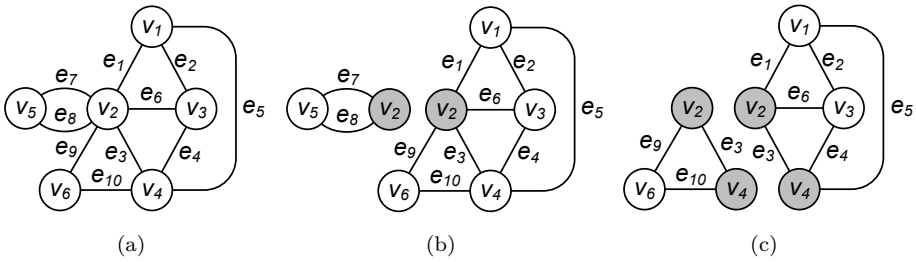


Figure 3.5. (a) An undirected graph, (b) the biconnected decomposition of (a), and (c) the triconnected decomposition of (b)

Connectivity-based decomposition starts with a connected graph. If a graph is disconnected, then it must be broken into connected subgraphs and the decomposition should proceed independently on each of the connected subgraphs. The graph in Figure 3.5(a) is connected. In a connected graph, there exists a path between every pair of vertices. However, the existence of a path is not guaranteed if one element gets removed from the graph (either a vertex or an edge). Vertex v_2 is the only cutvertex of the graph in Figure 3.5(a). If vertex v_2 gets removed, vertex v_5 gets disconnected from the rest of the graph. Therefore, the graph in Figure 3.5(a) is not biconnected. A connected graph can be decomposed into biconnected subgraphs by means of the *biconnected decomposition*, which computes the biconnected subgraphs induced by the removals of cutvertices of the graph.

For instance, Figure 3.5(b) shows two subgraphs of the graph in Figure 3.5(a) which are induced by the removal of its only cutvertex v_2.

Each of the subgraphs in Figure 3.5(b) has no separating set that is composed of a single vertex or single edge and, hence, the subgraphs are biconnected. One can proceed with the decomposition of these subgraphs into triconnected subgraphs. This can be accomplished by means of the *triconnected decomposition*, i.e., by removing separation pairs from the biconnected subgraphs. Because subgraph $(\{v_2, v_5\}, \{e_7, e_8\})$ in Figure 3.5(b) is complete, the decomposition should proceed only on one subgraph. Finally, Figure 3.5(c) shows two subgraphs induced by the removal of separation pair $\{v_2, v_4\}$. Both subgraphs in Figure 3.5(c) are complete and, thus, decomposition terminates.

The connectivity-based decomposition, as exemplified above, provides information on the compositional structure of a graph, i.e., subgraphs that the graph is composed of and the principles of the subgraphs composition in the graph. In the example above, we have first decided to decompose the graph based on its cutvertices and afterwards decomposed induced subgraphs based on separation pairs. Note that usually the decision on how to decompose the given (sub)graph cannot be determined uniquely. For instance, decompositions of the subgraph in Figure 3.5(b) can also be induced by removals of separation pairs $\{e_9, e_{10}\}$ or $\{v_2, e_{10}\}$. Every sequence of decisions along decomposition of a graph results in a unique graph decomposition strategy which allows one to observe unique structural characteristics of the graph.

3.2.3. The Tree of the Triconnected Components

A graph that is not connected can be uniquely partitioned into *connected components*, i.e., maximal connected subgraphs. A connected graph that is not biconnected can be uniquely decomposed into *biconnected components*, i.e., maximal biconnected subgraphs. The fragments of a TTG are closely related to its *triconnected components*, which was pointed out by Robert Endre Tarjan and Jacobo Valdes [124]. Because of this relationship, we are interested to decompose biconnected graphs into unique triconnected components. This relationship is crucial for the RPST computation that will be proposed in Section 3.3.1 and Section 3.3.2. Therefore, this section is devoted to the detailed discussion of the triconnected decomposition of biconnected graphs.

Any biconnected graph can be a subject to the triconnected decomposition. However, as it will become evident later, when discovering fragments in a TTG it is convenient to work with its completed version. The *completed version* of a TTG G, denoted by $C(G)$, is the undirected graph that results from ignoring the direction of all the edges of G and adding an additional edge between the source and the sink. The additional edge in the completed version of G is called the *return edge* of $C(G)$.

The TTG in Figure 3.1(b) is connected, but not biconnected; the nodes u, x, y, and z are all separation points. Figure 3.6 shows the completed version $C(G)$ of the TTG in Figure 3.1(b), where r is the return edge drawn with a dash-dotted

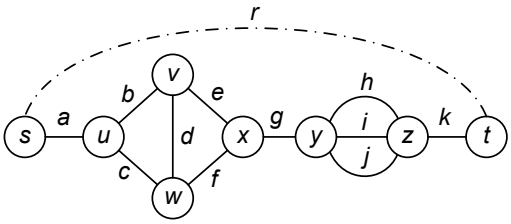
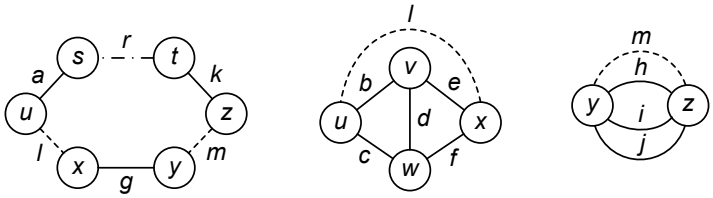


Figure 3.6. The completed version of the TTG in Figure 3.1(b)

line. The completed version is biconnected but not triconnected; $\{u, x\}$ and $\{x, z\}$ are two of many separation pairs of $C(G)$.

Fragments of a TTG are strongly related to the separation pairs of its completed version. Note that the entry-exit pair $\{u, x\}$ of fragment $R1$ in Figure 3.3(a) is also a separation pair of its completed version in Figure 3.6. In fact, each entry-exit pair of a non-trivial fragment of a TTG G is a separation pair of $C(G)$.

Let G be a biconnected multi-graph and u, v be two nodes of G. A *separation class w.r.t.* u, v is a maximal set S of edges such that any two edges in S are connected by a path that visits neither u nor v except as a start or end point. If there is a partition of all edges of G into two sets E_0, E_1 such that both sets contain more than one edge and each separation class w.r.t. u, v is contained in either of these sets, we call $\{u, v\}$ a *split pair*. We can then *split* the graph into two parts w.r.t. the parameters E_0, E_1 and u, v: To this end, we add a fresh edge e between u and v to the graph, which is called a *virtual edge*. The graphs formed by the sets $E_0 \cup \{e\}$ and $E_1 \cup \{e\}$ are the obtained *split graphs* of the performed *split operation*. A virtual edge is visualized by a dashed line.



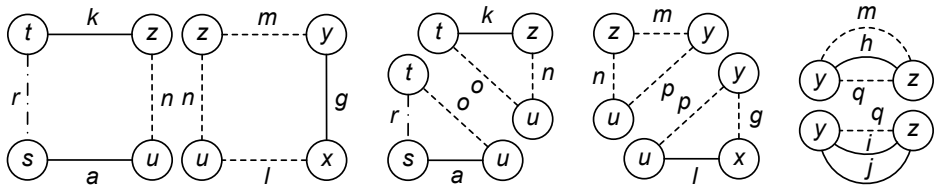
(a) (b) (c)

Figure 3.7. The triconnected components of the graph in Figure 3.6: (a) a polygon, (b) a rigid, and (c) a bond component

For an example of a split operation, consider the hexagon in Figure 3.7(a). Note that it already contains virtual edges, which are the result of previous splits. The hexagon can be split along the split pair u, z using the sets $E_1 = \{a, r, k\}, E_2 = \{l, g, m\}$. This results in two tetragons, which are shown in Figure 3.8(a).

It may be possible to split the obtained split graphs further, i.e., into smaller split graphs, possibly w.r.t. a different split pair. A split graph is called a *split component* if it cannot be split further. Special split graphs are *polygons* and *bonds*. A *polygon* is a graph that has $k \geq 3$ nodes and k edges such that all nodes

and edges are contained in a cycle, see Figure 3.7(a). A *bond* consists of 2 nodes and $k \geq 2$ edges between them, see Figure 3.7(c). Each split component is either a *triangle*, i.e., a polygon with three nodes, a *triple bond*, i.e., a bond with three edges, or a *simple* triconnected graph, where *simple* means that no pair of nodes is connected by more than one edge [54]. If a split component is the latter, we also call it a *rigid* component. Figure 3.7(b) shows an example of a rigid component, whereas the split graphs shown in Figure 3.7(a) and Figure 3.7(c) are *not* split components as they can be split further.



(a) (b) (c) (d)

Figure 3.8. (a) A split of a hexagon from Figure 3.7(a), (b),(c) splits of a tetragon, (d) a split of a bond

The set of split components that can be derived from a biconnected multi-graph is not unique. To see that, we consider polygons and bonds. For instance, a tetragon, see Figure 3.8(a), can be split along a diagonal into two split graphs. Depending on the choice of the diagonal, two different sets of split components are obtained. Figure 3.8(b) shows one of the two possibilities for splitting the tetragon given on the left in Figure 3.8(a). Similarly, a bond with more than three edges, see Figure 3.7(c), can be split into two bonds in several ways, depending on the choice of E_1 and E_2. One possibility to split the bond from Figure 3.7(c) is shown in Figure 3.8(d). A set of split components for the graph in Figure 3.6 is given by the graphs in Figures 3.7(b), 3.8(b), 3.8(c), and 3.8(d).

The inverse of a split operation is called a *merge operation*. Two split graphs formed by edges E_0 and E_1, respectively, that share a virtual edge e between a pair u, v of nodes can be merged, which results in the graph formed by the set $(E_0 \cup E_1) \setminus \{e\}$ of edges. If we start with a set of split components of G and then iteratively merge a polygon with a polygon and a bond with a bond until no more such merging is possible, we obtain the unique *triconnected components* of G. Because a merge operation is the inverse of a split operation, we can also obtain the triconnected components by suitable split operations only.

Let $\mathcal{C}$ be a *split graph decomposition* of G, i.e., a set of split graphs recursively derived from G. A polygon $P \in \mathcal{C}$ is *maximal* w.r.t. $\mathcal{C}$ if there is no other polygon in $\mathcal{C}$ that shares a virtual edge with P. A bond $B \in \mathcal{C}$ is *maximal* w.r.t. $\mathcal{C}$ if there is no other bond in $\mathcal{C}$ that shares a virtual edge with B. $\mathcal{C}$ is a set of the *triconnected components* of G if each member of $\mathcal{C}$ is either a maximal polygon, a maximal bond, or a rigid split component. The set of the triconnected components of G exists and is unique [54].

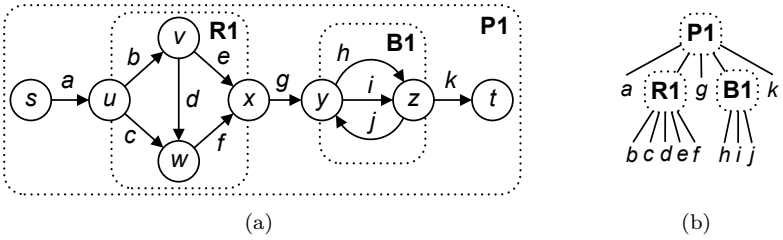
The graphs in Figure 3.8(c) can be merged along the virtual edge p. The obtained tetragon can be merged with the triangles in Figure 3.8(b) along the virtual edges

n and o to obtain the maximal polygon from Figure 3.7(a). Figures 3.7(a), 3.7(b), and 3.7(c) show all the triconnected components of the graph from Figure 3.6: Figure 3.7(c) is a maximal bond, which is obtained by merging the bonds in Figure 3.8(d), and Figure 3.7(b) is a rigid component.

Any split graph decomposition can be arranged in a tree: The tree nodes are the split graphs. Two split graphs are connected in a tree if they share a virtual edge. The root of the tree is the split graph that contains the return edge. The *tree of the triconnected components* of G is the tree derived in this way from its triconnected components.

Definition 3.6 (The Tree of the Triconnected Components).
The *Tree of the Triconnected Components* (TTC) of a biconnected graph G is the set of all triconnected components of G. ⌟

Let C be a triconnected component of graph G. Let F be the set of all edges of G that appear in C or some descendant of C in the tree of the triconnected components. The graph formed by F is called the *triconnected component subgraph* derived from C.



(a) (b)

Figure 3.9. (a) A TTG and its triconnected component subgraphs, (b) the tree of the triconnected components of (a)

Figure 3.9 shows the tree of the triconnected components of the TTG in Figure 3.1(b). In Figure 3.9(a), the triconnected component subgraphs are visualized; they correspond to the triconnected components from Figure 3.7. Each triconnected component subgraph is formed by edges enclosed in or intersecting a region with the dotted border, e.g., all the graph edges for $P1$ are derived from the component given in Figure 3.7(a). Figure 3.9(b) arranges the triconnected components in a tree. The root of the tree, i.e., node $P1$, corresponds to the triconnected component that contains the return edge r. Note the difference between the tree of the triconnected components in Figure 3.9(b) and the RPST in Figure 3.3(b). The names of the subgraphs hint at the types of the triconnected components, i.e., $P1$ is a polygon, $B1$ is a bond, and $R1$ is a rigid.

3.3. Simplified Computation of the RPST

This section proposes an algorithm for the computation of the RPST. The algorithm is simplified if compared with the original one [145, 146]. First, Section 3.3.1

discusses the RPST of TTGs in which every node has at most one incoming or at most one outgoing edge. Such TTGs are common in practice and are referred to as normalized TTGs. Afterwards, Section 3.3.2 addresses the general case of the RPST computation of any TTG whose completed version is biconnected.

3.3.1. The RPST of Normalized TTGs

We call a TTG *normalized*, if every node has at most one incoming or at most one outgoing edge. In this section, we show that for normalized TTGs, the RPST computation reduces to computing the tree of the triconnected components. In other words, each canonical fragment corresponds to a triconnected component subgraph and each triconnected component subgraph corresponds to a canonical fragment.

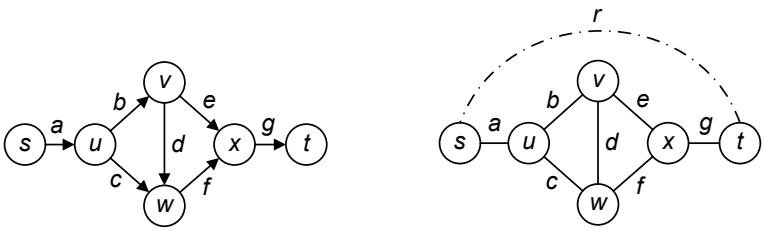


Figure 3.10. (a) A normalized TTG and (b) its completed version

Let $C(G)$ be the completed version of a TTG. A pair $\{x, y\}$ of nodes is called a *boundary pair*, if there are at least two separation classes w.r.t. $\{x, y\}$. A separation class is *proper*, if it does not contain the return edge. Figure 3.10 shows a normalized TTG and its completed version. The TTG is formed by a subset of edges of the workflow graph from Figure 3.1(b). The boundary pair $\{u, x\}$ in Figure 3.10(b) generates two separation classes. The first contains the edges b, c, d, e, f and is, therefore, proper; whereas the second contains all other edges of the graph, including the return edge r, and is therefore not proper. Fragments are strongly related to proper separation classes. To describe that relationship, we introduce the notion of a *separation component*.

Definition 3.7 (Separation component).
Let $\{x, y\}$ be a boundary pair of $C(G)$. A *separation component* w.r.t. $\{x, y\}$ is the union of one or more proper separation classes w.r.t. $\{x, y\}$. ⌟

The bond from Figure 3.7(c) without the virtual edge m is a separation component w.r.t. $\{y, z\}$ of the completed version of the TTG from Figure 3.6. It is the union of the three proper separation classes: $\{h\}$, $\{i\}$, and $\{j\}$.

We know that the entry-exit pair $\{x, y\}$ of a fragment is a boundary pair of G and that the fragment is a *separation component* w.r.t. $\{x, y\}$ [146]. Furthermore, it follows from the construction of the triconnected components that each triconnected component subgraph is a separation component. We show now that every triconnected component subgraph of a normalized TTG is a fragment.

Lemma 3.8: *Let F be a triconnected component subgraph of a normalized TTG. F is a fragment.* ⋆

Proof. Every triconnected component subgraph of a normalized TTG has two boundary nodes; they form a boundary pair. We want to show that one of the boundary nodes is an entry and the other one is an exit of the subgraph. First, we show that a boundary node of a triconnected component subgraph is either an entry or an exit. A boundary node of a subgraph is an entry or an exit, if either all incoming or all outgoing edges of the node are part of the subgraph or are outside the subgraph (Definition 3.2). For every node of a normalized TTG it holds that either a set of all incoming edges or a set of all outgoing edges contains one edge. The configuration of this one edge, i.e., either it belongs to the subgraph or not, defines the configuration of the whole set. Therefore, any boundary node of a triconnected component subgraph is either an entry or an exit. Finally, the fact that only one arrangement of boundary nodes is possible, i.e., one of the nodes is an entry and the other one is an exit, follows from the definition of a TTG (Definition 3.1). The arrangement of two entry or two exit boundary nodes violates the requirement that each node of a TTG lies on a path from the source to the sink. □

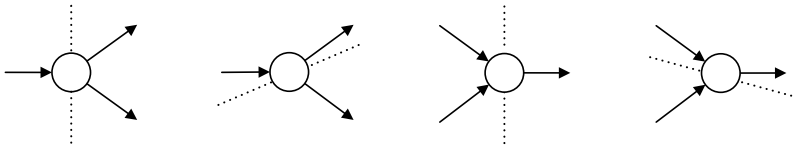


Figure 3.11. Different configurations of a boundary node with three incident edges

Figure 3.11 exemplifies Lemma 3.8; it shows all possible configurations of a boundary node with three incident edges. The dotted lines separate edges on internal and external subgraph edges. Regardless of the configuration, each node in the figure can be classified as either an entry or an exit. For normalized TTGs, we can extend the observation of Lemma 3.8 to a full characterization of fragments in terms of separation components.

Lemma 3.9: *Let F be a set of edges of a normalized TTG. F is a separation component, if and only if F is a fragment.* ⋆

Proof. We prove each direction of the statement separately.

(⇒) Let $\{u, v\}$ be the boundary pair of F and let e be an edge in F. As the return edge is not in F, it is in a different separation class w.r.t. $\{u, v\}$ than e. Consider a simple directed path from the source to the sink of the graph that contains e. It follows that the path contains one of the nodes $\{u, v\}$ before e and one after e; otherwise the separation class of e would contain the return edge. Let, without loss of generality, u be the former

node and v the latter. It follows that u has an incoming edge outside F and an outgoing edge inside F, and v has an incoming edge inside F and an outgoing edge outside F. Based on the assumption that the TTG is normalized, it is now straightforward to establish that u is an entry and v is an exit of F. Furthermore, there is no other boundary node besides u and v because that would contradict the definition of a separation class. Hence, F is a fragment.

($\Leftarrow$) See Theorem 2 in [146]. □

It turns out that the set of triconnected component subgraphs of a normalized TTG is exactly the set of all its canonical fragments and, thus, is the RPST of the TTG. Before we prove the statement, we give two auxiliary lemmas which also by themselves deliver interesting insights into separation components of a normalized TTG and their relations.

Lemma 3.10: *If F is a separation component and F' a triconnected component subgraph, then F and F' do not overlap.* ⋆

Proof. If F contains only a single edge or the entire graph, the claim is trivial. Otherwise F can be split off from the main graph into a split graph. We continue the decomposition until we reach a set of split components. Those can be arranged in a tree (of split components) as described above. F corresponds to a subgraph of this tree, i.e., a subtree represents exactly the edges of F. On the other hand, F' also corresponds to a subtree of the tree of split components because the triconnected components are obtained by merging split components, i.e., by collapsing parts of the tree of split components. Since F and F' both correspond to subtrees of the same tree, they do not overlap. □

It follows from Lemma 3.10 that triconnected component subgraphs do not overlap. We show now that for a separation component which is strictly contained in a triconnected component subgraph, there always exists another separation component contained in the same triconnected component subgraph that overlaps with it.

Lemma 3.11: *If F is a separation component that is not a triconnected component subgraph, then there exists a separation component F', such that F and F' overlap.* ⋆

Proof. Consider a split graph decomposition that contains F. If F is not a triconnected component subgraph, then F and the parent of F are either bonds w.r.t. the same boundary pair or polygons. In both cases, it is easy to display a bond or polygon, respectively, that overlaps with F. □

We are now ready to prove the main proposition of this section.

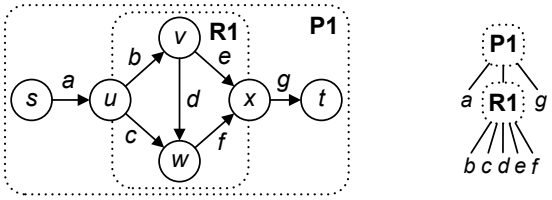
Theorem 3.12. *Let F be a set of edges of a normalized TTG. F is a canonical fragment, if and only if F is a triconnected component subgraph.* ⋆

Proof. We prove each direction of the statement separately.

(⇒) Let F be a canonical fragment. We want to show that F is a triconnected component subgraph. Because of Lemma 3.9, F is a separation component. If F is not a triconnected component subgraph, then there exists, because of Lemma 3.11, a separation component F' that overlaps with F. Because of Lemma 3.9, F' is a fragment, which contradicts F being canonical.

(⇐) Let F be a triconnected component subgraph. We want to show that F is a canonical fragment. Because of Lemma 3.9, F is a fragment. Let F' be any fragment. Because of Lemma 3.9, F' is a separation component. Because of Lemma 3.10, F and F' do not overlap. Hence, F is a canonical fragment.□

For normalized TTGs, Theorem 3.12 implies that the tree of the triconnected components and the RPST coincide, i.e., both deliver the same decomposition on the set of edges of the TTG.



(a) (b)

Figure 3.12. (a) A TTG and its triconnected component subgraphs, (b) the RPST of (a)

Figure 3.12(a) shows the triconnected component subgraphs of the normalized TTG in Figure 3.10(a). The triconnected component subgraphs are also all the canonical fragments of the TTG. Therefore, the RPST of the workflow graph from Figure 3.12(a), which is given in Figure 3.12(b), can be computed by constructing the tree of the triconnected components of the workflow graph.

3.3.2. The RPST of General TTGs

We now show how to compute the RPST of an arbitrary TTG whose completed version is biconnected. To do so, we normalize the TTG by *splitting* nodes that have more than one incoming and more than one outgoing edge into two nodes. We then compute the RPST of the resulting *normalized* TTG as in Section 3.3.1. Finally, we project the RPST of the normalized TTG onto the original one and obtain the RPST of the original TTG.

We start the discussion with the presentation of the node-splitting operation.

Definition 3.13 (Node-splitting).
Let $G = (V, E, \ell)$ be a directed multi-graph and $x \in V$ a node of G. A *splitting* of x is *applicable* if x has more than one incoming and more than one outgoing edge. The application results in a graph $G' = (V', E', \ell')$, where $V' = (V \setminus \{x\}) \cup \{*x, x*\}$, $E' = E \cup \{e\}$, where $*x$ and $x*$ are fresh nodes and e is a fresh edge, and ℓ' is such

that $\ell'(e) = (*x, x*)$. In addition, $f \in E, \ell(f) = (y, z)$ and $\ell'(f) = (y', z')$ implies that $y' = x*$ if $y = x$, otherwise $y' = y$; and $z' = *x$ if $z = x$, otherwise $z' = z$. ⌟

A single node-splitting is sketched in Figure 3.13(a). For instance, if the splitting is applied to node u of the graph from Figure 3.13(b), it results in the new graph given in Figure 3.13(c) with three fresh elements: nodes $*u$ and $u*$, and edge e. This is the only applicable splitting in the example. Hence, the resulting graph is normalized and we call it the *normalized version* of the TTG. The procedure can be formalized as follows. Splitting is applicable if and only if the graph is not normalized. It is not difficult to see that the order of different splittings does not influence the final result and, therefore, we indeed get a normal form by applying all applicable splittings in any order.

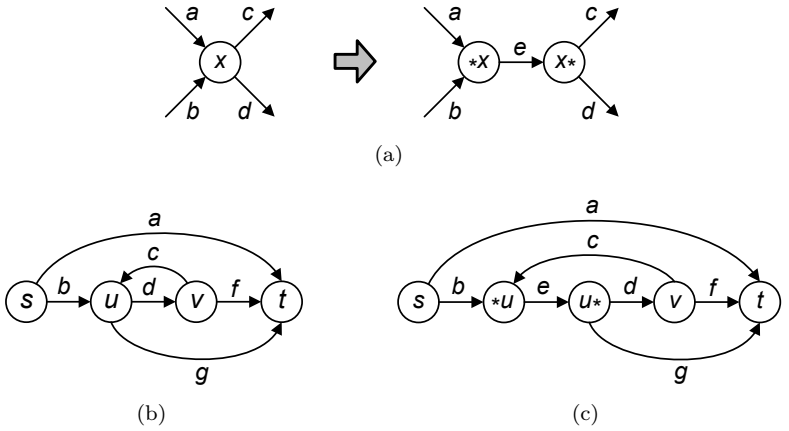


Figure 3.13. (a) Node-splitting, (b) a TTG, and (c) the normalized version of (b)

After normalization, we proceed by computing the tree of the triconnected components of the graph. As we know from Section 3.3.1, the tree coincides with the RPST of the normalized graph. This tree can be projected onto the original graph by deleting all the edges introduced during node-splittings. We will see later that this projection preserves the fragments. However,the deletion of the edges may result in fragments which have a single child fragment. This means that two different fragments of the normalized graph project onto the same fragment of the original graph. We thus clean the tree by deleting redundant occurrences of such fragments. Consequently, the only child fragment of a redundant fragment becomes a child of the parent of the redundant fragment, or the root of the tree if the redundant fragment has no parent. The result is the RPST of the original graph. Algorithm 1 details again the sequence of these steps.

We exemplify Algorithm 1 in Figure 3.14 and Figure 3.15 by computing the RPST of the TTG from Figure 3.14(a). Figure 3.14(a) shows the triconnected component subgraphs $P1$ and $B1$ of the TTG, whereas Figure 3.14(b) shows the corresponding tree of the triconnected components. The TTG is not normalized: Nodes y and z are incident with multiple incoming and multiple outgoing edges;

Algorithm 1: Simplified computation of the RPST

Input: A TTG $G = (V, E, \ell)$, such that $C(G)$ is biconnected
Output: The RPST of G

1 **if** $C(G)$ *is not biconnected* **then** FAIL
2 Construct $G' = (V', E', \ell')$ – the normalized version of G
3 Compute T' – the TTC of G'
4 Construct $T - T'$ without trivial fragments in $E' \backslash E$
5 Construct $R - T$ without redundant fragments
6 **return** R // `the RPST of` G

observe that all the triconnected component subgraphs of the TTG are fragments. Figure 3.14(c) shows the normalized version of the TTG from Figure 3.14(a); it is obtained by splitting nodes y and z, in any order (line 2, Algorithm 1). The normalization introduces edges l and m to the TTG. The tree of the triconnected components of the normalized version consists of four triconnected components: $P1$, $B1$, $P2$, and $B2$ shown in Figure 3.14(c). It follows from Lemma 3.9 that they are all fragments.

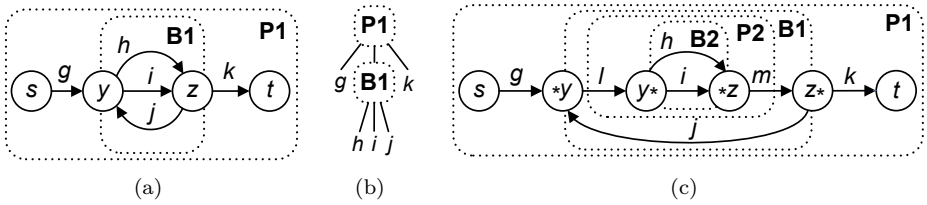


Figure 3.14. (a) A TTG and its triconnected component subgraphs, (b) the tree of the triconnected components of (a), and (c) the normalized version of (a) and its triconnected component subgraphs

Figure 3.15(a) shows the tree of the triconnected components of the normalized version from Figure 3.14(c) (line 3, Algorithm 1). Because of Theorem 3.12, the tree is the RPST of the normalized version. In Figure 3.15(b), one can see the RPST without trivial fragments, which correspond to the edges l and m (line 4). Note that $P2$ now specifies the same set of edges of the TTG as $B2$. Therefore, we omit $P2$, which is redundant, to obtain the tree given in Figure 3.15(c) (line 5). This tree is the RPST of the original TTG from Figure 3.14(a). Figure 3.15(d) visualizes the TTG again together with its canonical fragments. Please note that Algorithm 1, in comparison with the triconnected decomposition shown in Figure 3.14(a) and Figure 3.14(b), additionally discovered canonical fragment $B2$. $P1$, $B1$, and $B2$ are all the canonical fragments of the TTG.

To show that we indeed obtain the RPST of the original graph, we have to show that (i) each canonical fragment of the normalized version projects onto a canonical fragment of the original graph or onto the empty set, and (ii) for each

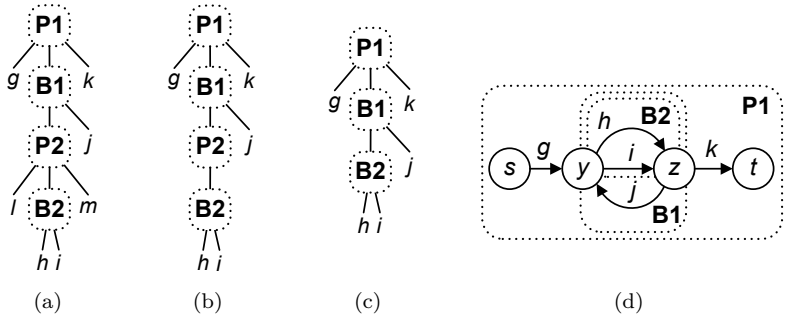


Figure 3.15. (a) The tree of the triconnected components of the TTG from Figure 3.14(c), (b) the tree from (a) without the fresh edges l and m, (c) the RPST of the TTG from Figure 3.14(a), and (d) the TTG from Figure 3.14(a) and its canonical fragments

canonical fragment of the original graph, there is a canonical fragment of the normalized version that is projected onto it. We establish these properties for a single node-splitting step. The claim then follows by induction.

Consider a single node-splitting step transforming a graph G into G', let x be the node that is split into nodes $*x$ and $x*$, and let e be the edge that is added between $*x$ and $x*$. We define the following mappings for the next lemma:

1. A mapping ψ maps a set F of edges of G' to a set $\psi(F)$ of edges of G by $\psi(F) = F \setminus \{e\}$.
2. A mapping ϕ maps a set of edges H of G to a set $\phi(H)$ of edges of G' by $\phi(H) = H \cup \{e\}$ if H has an incoming edge to x as well as an outgoing edge from x, and otherwise $\phi(H) = H$.

Now, we claim:

Lemma 3.14: *Let ϕ, ψ and e be defined as above. We have:*

1. *If $F \neq \{e\}$ is a fragment of G', then $\psi(F)$ is a fragment of G.*
2. *If H is a fragment of G, then $\phi(H)$ is a fragment of G'.*
3. *If $F \neq \{e\}$ is a canonical fragment of G', then $\psi(F)$ is a canonical fragment of G.*
4. *If H is a canonical fragment of G, then there exists a canonical fragment F of G' such that $\psi(F) = H$.* ⋆

The proof of Lemma 3.14 is in Appendix A.1. Lemma 3.14 and the fact that each step in Algorithm 1 can be computed in linear time allow us to conclude:

Theorem 3.15. *Algorithm 1 computes the RPST of a TTG whose completed version is biconnected in linear time.* ⋆

3.4. Generalization of the Refined Process Structure Tree

So far, the RPST decomposition is restricted to TTGs whose completed version is biconnected. In practice this is often not sufficient, as behavioral models may

violate biconnectedness assumption, consider Figure 3.16(a), may have multiple sources and/or sinks, see Figure 3.16(b), or even be disconnected. Note that modeling languages such as BPMN and EPC do not impose any structural limitations in these respects. In Figure 3.16(a), node u is a separation point of the completed version of the graph; its deletion separates the node labeled A from the rest of the graph. Hence, the completed version is not biconnected. In fact, a test of the SAP reference model [19], a collection of industrial process models given as EPCs, showed that more than 80 percent of the models violate one of the restrictions.

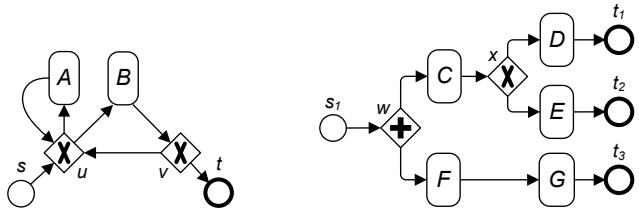


Figure 3.16. A process model (a) whose completed version is *not* biconnected, (b) has multiple sinks

In this section, we discuss a way to decompose any MTG. The results of this section were originally proposed in the thesis of Jussi Vanhatalo [144]. In the following, we summarize these results and show how they relate to the simplified technique for the RPST computation, which was described in the previous section. We start by decomposing arbitrary TTGs.

3.4.1. The Refined Process Structure Tree of TTGs

Figure 3.17(a) shows the TTG that corresponds to the process model in Figure 3.16(a). As we explained above, its completed version is not biconnected because node u is a separation point. Note that u has multiple incoming as well as multiple outgoing edges. Every separation point has this property:

Lemma 3.16: *Let G be a TTG. Every separation point of $C(G)$ has more than one incoming and more than one outgoing edge in G.* ⋆

Proof. A source s and a sink t of G are in the same biconnected component of $C(G)$ as they are connected in G and, therefore, biconnected in $C(G)$ after introducing the return edge. Moreover, it is easy to see that $C(G)$ is connected without s or t and, hence, s and t are not separation points of $C(G)$. Let x, without loss of generality, be some separation point of $C(G)$ that results in a set B of biconnected components. Let $b \in B$, without loss of generality, be a biconnected component induced by x that does not contain s and t. Assume y is a node which belongs to b. As every node of G is on a path from s to t, then x is on every path from s to y and from y to t. A path from s to y implies that x has an incoming edge that does not belong to b and an outgoing edge that belongs to b. A path

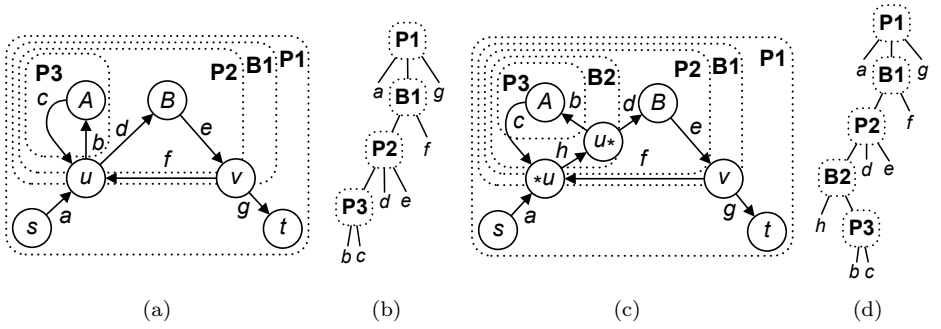


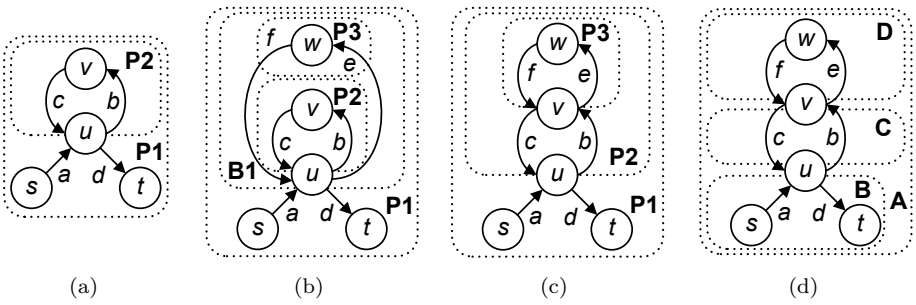
Figure 3.17. (a) A TTG whose completed version is *not* biconnected, (b) the RPST of (a), (c) the normalization of (a), and (d) the RPST of (c)

from y to t implies that x has an incoming edge that belongs to b and an outgoing edge that does not belong to b. Hence, the claim holds.

If b consists of a single edge, it is an incoming and an outgoing edge of x. Every path from s to t through x also contains two edges incident with x, an incoming and an outgoing, which do not belong to b. Hence, the claim holds. □

It follows that the completed version of the normalization of G is biconnected. Therefore, we can apply Algorithm 1 from Section 3.3.2 to decompose an arbitrary TTG. We call the resulting decomposition of G the *RPST* of G. This is a generalization of the previous definition because if $C(G)$ is already biconnected, we get the RPST as defined previously. Note that we obtain the same result by splitting only the separation points of G, computing the RPST of the resulting graph G' (in any way), and then projecting the RPST of G' onto G. As the normalized version and its RPST are unique, it then follows from the construction that the RPST of an arbitrary TTG is unique.

Figure 3.17 shows the RPST of the example, as well as the way in which it is obtained. Again, the RPST of the original graph is obtained by deleting the edge h, which was generated in the node-splitting, and afterwards removing the redundant fragment $B2$.



(a) (b) (c) (d)

Figure 3.18. (a)–(c) The RPST of a TTG, and (d) Valdes's parse tree of the TTG in (c)

Figures 3.18(a), 3.18(b), and 3.18(c) show more examples of decompositions of TTGs whose completed versions are not biconnected. Every subgraph obtained has either exactly two boundary nodes, one entry and one exit, or exactly one boundary node, which is *bidirectional*. Let G be a TTG and F be a connected subgraph of G. A boundary node u of F is *bidirectional* if there exist an incoming and an outgoing edge of u inside F, and there exist an incoming and an outgoing edge of u outside F. Note that control flow can both enter and exit F through u.

Jacobo Valdes [127] has proposed an alternative way to decompose an arbitrary TTG G. He proposed to first compute the *biconnected components* of $C(G)$ and then further decompose each biconnected component into its triconnected components. If we adapt this idea and compute the RPST of each biconnected component of $C(G)$, we obtain a root component that contains all biconnected components as children, which in turn have their RPSTs as subtrees. The result for the graph from Figure 3.18(c) is shown in Figure 3.18(d), which is different from the decomposition we propose. Note that the result has a component that has more than two boundary nodes, e.g., B, and another one having two boundary nodes that are both bidirectional, e.g., C. Unlike our decomposition, the decomposition in Figure 3.18(d) does not reflect the fact that the component containing node w depends on the component that is entered through node u.

3.4.2. The Refined Process Structure Tree of MTGs

To decompose an arbitrary MTG, we "normalize" an MTG into a TTG by constructing a unique source and a unique sink as follows.

Definition 3.17 (Normalized MTG).
Let G be an MTG. We construct a graph G' from G as follows.

1. If G has more than one source, a new source s is added and for each source node u of G, an edge from s to u is added.
2. If G has more than one sink, a new sink t is added and for each sink node v of G, an edge from v to t is added.

G' is a TTG, which we call the *TTG version* of G. The *normalized version* G^* of G is the normalized version of G'. ⌟

By normalizing an MTG, we again obtain a TTG whose completed version is biconnected. The normalized version can be decomposed with the RPST, and the decomposition can be projected onto the original MTG through Algorithm 1. The result that is obtained from applying Algorithm 1 to the normalized version of an MTG G is called the *RPST* of G. The RPST of an MTG is unique.

Figure 3.19 shows (a) an MTG G, (b) the RPST of G, (c) the TTG version G' of G, and (d) the RPST of G'. The RPST of G is derived from the RPST of G' with Algorithm 1.

Note that for an MTG, the subgraphs formed by the decomposition may have more than two boundary nodes. For example, subgraph $B1$ in Figure 3.19(a) has two sources u and v as entries, and an exit w. Subgraph $B2$ has an entry w, and three sinks as exits. Subgraph $P1$ two sources as entries, and three sinks as exits.

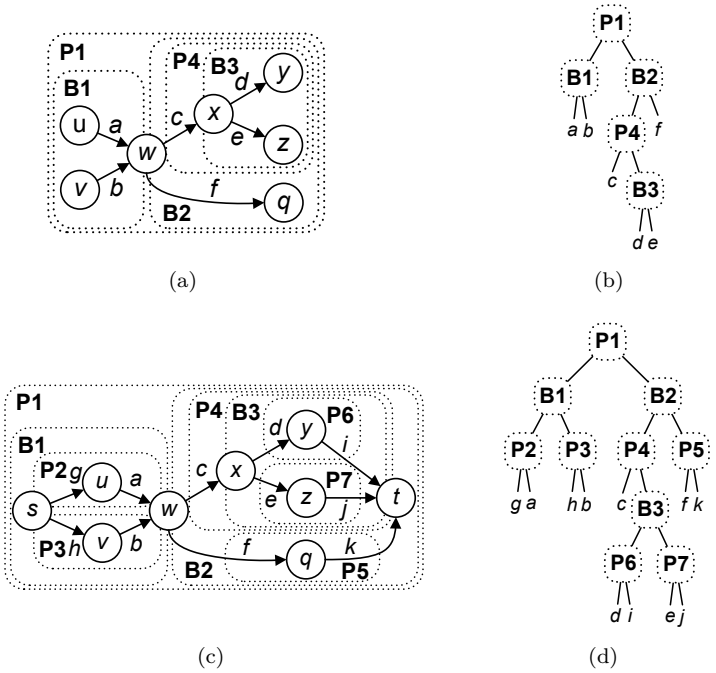


Figure 3.19. (a) An MTG G, (b) the RPST of G, (c) the TTG version G' of G, and (d) the RPST of G'

An RPST-formed subgraph is not necessarily a connected subgraph of an MTG. If an MTG is disconnected, the root fragment of its RPST is a union of the connected components of the MTG. For example, Figure 3.20 shows an example of (a) a disconnected MTG G, (b) the RPST of G, (c) the TTG (and normalized) version G^* of G, and (d) the RPST of G^*. Note that every connected component of the MTG always becomes a separate component of the RPST decomposition. In the example, the connected components of the MTG are fragments $P1$ and $P2$, see in Figure 3.20(a); $P1$ and $P2$ are parts of root fragment $B1$.

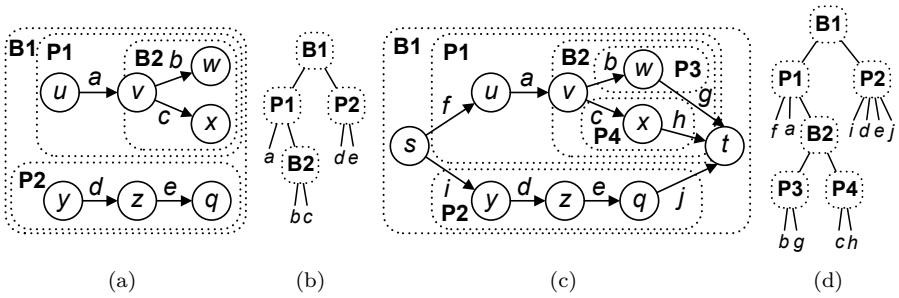


Figure 3.20. (a) A disconnected MTG G, (b) the RPST of G, (c) the TTG version G^* of G, and (d) the RPST of G^*

3.5. Bibliographical Notes and Conclusion

In this section, we mention important works which fueled research on parsing of (work)flow graphs, position the results of this chapter within related works, and draw conclusions. We start with bibliographical notes on related works:

- The connectivity properties are the basic properties of graphs and are useful when testing whether a graph is planar or when determining if two graphs are isomorphic. John Hopcroft and Robert Endre Tarjan (1973) developed an optimal (to within a constant factor) algorithm for dividing a graph into triconnected components [54]. The algorithm is based on the depth-first search of graphs and requires $O(V + E)$ time and space to examine a graph with V vertices and E edges.
- Robert Endre Tarjan and Jacobo Valdes (1980) used triconnected components for structural analysis of biconnected flow graphs [124]. The triconnected components of the undirected version of a flow graph are shown to be useful for discovering structural information of directed flow graphs. The triconnected components can be discovered efficiently and form a hierarchy of SESE fragments of a flow graph.
- Giuseppe Di Battista and Roberto Tamassia (1990) introduced SPQR-trees [5] – a data structure which represents decomposition of a biconnected graph with respect to its triconnected components. Essentially, SPQR-trees are the parse trees of [124]. The authors showed the usefulness of SPQR-trees for various on-line graph algorithms[1], e.g., transitive closure, planarity testing, and minimum spanning tree [5]. In particular, the authors proposed an efficient solution to the problem of on-line maintenance of the triconnected components of a graph [6].
- Richard Johnson et al. (1994) proposed a program structure tree (PST), a hierarchical representation of program structure based on single edge entry and single edge exit regions [63, 62]. The PST can be computed in $O(E)$ time for an arbitrary flow graph, where E is the number of edges in the graph. The disadvantage of the PST is that it exploits the notion of a SESE fragment based on edge entries and exits only. Thus, the PST does not capture those SESE fragments which are based on vertex entries and exits.
- Chun Ouyang et al. (2006) used parsing to translate BPMN diagrams into BPEL processes [98, 99]. The employed notion of a fragment is similar to the notion of a region in [63]. However, the developed parsing algorithm is nondeterministic, i.e., the parse tree is not unique for a given diagram.
- Carsten Gutwenger and Petra Mutzel (2001) shared their practical experience on linear time computation of the triconnected components of biconnected graphs [48]. They have identified and corrected the faulty parts of the

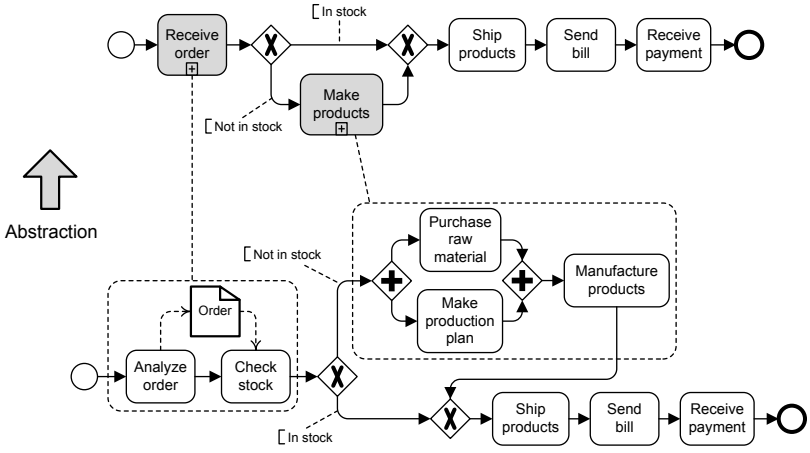
[1] An on-line graph problem deals with dynamic graphs, i.e., it is allowed to query and update a graph, such that each operation is completed before the next one can be applied, and future operations are not known in advance. Given a solution to a problem for some graph configuration the challenge is to update the solution, rather than to compute it from scratch, for the modified graph.

algorithm in [54] and applied the resulting algorithm to the computation of SPQR-trees. The implementation is publically available.

- Jussi Vanhatalo et al. (2008) introduced the Refined Process Structure Tree [145, 146, 144]. Given a workflow graph, the RPST is unique, modular, and is finer grained than any other known parse tree, i.e., it discovers more SESE fragments than any other technique. In fact, the RPST captures all canonical fragments of a workflow graph which, in turn, represent all SESE fragments of the graph. The RPST can be computed for an arbitrary MTG [144].

This chapter discussed the RPST. The main contribution of this chapter, as compared to the original technique proposed in [145, 146], is the new simplified algorithm for computation of the RPST of a TTG whose completed version is biconnected. This simplified algorithm can be employed in a straightforward way as a subroutine for computation of the RPST of an arbitrary MTG. Both algorithms, the original and the simplified one, allow for an efficient computation of the RPST. However, they provide different structural characterizations of canonical fragments. The original technique characterizes some fragments of a workflow graph as compositions of special bond types [144], whereas the new approach shows a tight relation of these fragments with the triconnected components of the normalized version of the graph. Both characterizations contribute to a better understanding of the essential nature of canonical fragments of a workflow graph.

4. Abstraction



Companies use behavioral models to represent their working procedures in order to deploy services to markets, to analyze them, and to improve upon them. Competitive markets necessitate complex procedures, which lead to large behavioral models with sophisticated structures. Real world behavioral models can often incorporate hundreds of modeling constructs. While a large degree of detail complicates the comprehension of the model, it is essential for many analysis tasks. This chapter presents a technique to abstract, i.e., to simplify behavioral models. Given a detailed model, we introduce abstraction rules which generalize model fragments in order to bring the model to a higher abstraction level. The approach is suited for the abstraction of large behavioral models in order to aid model comprehension, as well as decomposing problems of model analysis. The work is based on the notion of the parse tree, which was discussed in Chapter 3.

The materials reported in this chapter are published in [110, 111, 105, 112].

4.1. About Abstraction of Behavioral Models

Behavioral models are developed for different purposes: to communicate a message, to share knowledge or a vision, as a starting point for re-designing or optimization, as precise instructions for execution, etc. The goal of a behavioral model is to capture working procedures at a level of detail appropriate to fulfill its envisioned purpose. Often, achievement of such a goal results in complex, "wallpaper-like" models that tend to capture every minor detail and exceptional case that might occur; the core process logic becomes hidden in numerous modeling constructs.

The desired level of model granularity also depends on a stakeholder working with a model and a current task. If we talk about business process models [155], top level company management appreciates coarse grained process descriptions that allow fast and correct business decisions. At the same time, employees who directly execute processes value the granular specifications of their daily job. There is a dilemma: On the one hand, too much detail hampers the understanding of the model. On the other hand, this level of detail might be required for the purpose of the model. Thus, it might be often the case that a company ends up with maintaining several models of one business process.

There are two approaches to address the problem: (i) Either different models serving different purposes are developed independently. (ii) Alternatively, different models, catering to different needs, are generated from a detailed original model. If the former approach is followed, consistency of the models is a severe problem [153]. Changes on one level need to be reflected on other levels as well, which is often done manually. Experience shows that due to model evolution on different levels of detail, the models become inconsistent quite soon. Therefore, we opt for the latter approach: We generate different models from a given detailed model by applying transformation rules. These rules abstract from the details of a model and provide abstract models which are tailored to particular needs. At the same time, any evolutionary changes are taken into account, since effectively there is only one model, and the others are generated from it on demand.

Abstraction is generalization that reduces undesired details in order to retain only essential information about an entity or a phenomenon. Essential information is the information required by a certain stakeholder to fulfill his/her tasks. The task of abstraction is to tell significant model elements from insignificant ones and to reduce the latter.

We propose an abstraction methodology for behavioral models that can be summarized as follows. As input we assume to possess a detailed behavioral model. Afterwards, a number of abstractions, viz. *abstraction steps*, are performed on the initial model. Conceptually, each abstraction step is a function that takes a behavioral model as input and produces a behavioral model as output. In the resulting model, the initial fragment of the model gets replaced with its generalized version. Thus, each individual abstraction step hides details and brings the model to a higher abstraction level. If applied separately, abstraction steps do not provide much value to an end user. Rather, it is of interest to study how individual abstraction steps can be combined together and afterwards controlled in order to deliver the desired abstraction level.

Fundamentally, a technique for abstraction of behavioral models deals with finding answers to two questions of *what* and *how*:

- What parts of a behavioral model are of low significance?
- How to transform a behavioral model so that insignificant parts get reduced?

Answers to both questions should address the current abstraction use case, i.e., the resulting abstract model must serve its purpose.

In this chapter, we propose one concrete instantiation of the above described principles for the abstraction of behavioral models. We found this configuration of a particular use when modularizing the structuring problem, the solution to which will be proposed in Part III. In the next section, we define a set of abstraction rules; the rules reuse the results of Chapter 3 on parsing of behavioral models and aim at answering the *how* question of the abstraction methodology. Afterwards, in Section 4.3, we propose an *abstraction slider* – a mechanism providing a user control over the abstraction. The slider answers the *what* question of the abstraction methodology. Section 4.4 proposes a technique for extracting abstract parts from behavioral models. We shall employ this technique in Part III in order to extract unstructured parts and to abstract from already structured parts of behavioral models. Finally, Section 4.5 draws conclusion.

4.2. The Triconnected Abstraction

This section answers the *how* question of the abstraction methodology for behavioral models. The answer is implemented by means of the triconnected abstraction technique. Given a behavioral model, the main idea of the triconnected abstraction technique is to interchange single-entry-single-exit fragments of the model with fresh abstract tasks of higher abstraction levels (coarse grained tasks). Every such exchange operation constitutes an abstraction step. An abstraction step is triggered by a task node – a node with at most one direct predecessor and at most one direct successor, which is insignificant for the purpose of the model and, hence, can be abstracted. In order to allow for a gradual abstraction experience, the triconnected abstraction technique substitutes the *smallest* SESE fragment that contains the insignificant task with a fresh abstract task that semantically corresponds to the behavior encoded in the SESE fragment. Note that the fresh task might again be considered insignificant and trigger the next abstraction step. The approach assumes the existence of an abstraction control mechanism which delivers a set of tasks (nodes) to be abstracted in the behavioral model.

The triconnected abstraction technique is based on the parsing technique from Chapter 3. First, we define individual abstraction rules, see Section 4.2.1. Afterwards, we combine the rules into the abstraction algorithm, see Section 4.2.2.

In the following, we assume that behavioral models are formalized as normalized TTGs, refer to Section 3.3.1. Note that subsequently proposed results can be trivially generalized to become applicable for all MTGs by following the RPST generalization principles from Section 3.3.2 and Section 3.4.

4.2.1. Abstraction Rules

In this section, we propose a set of abstraction rules. Each abstraction rule gets a task node of a behavioral model as input and defines: (i) a SESE fragment which must be abstracted and (ii) transformations to be applied in the behavioral model in order to implement the abstraction step. The starting point for rule definitions is the set of canonical fragments of the behavioral model, i.e., its RPST. Please note that in the following we shall exemplify abstraction rules using transformations of triconnected components. Recall that triconnected components of a normalized TTG define its RPST, see Theorem 3.12.

Given a task node, it can be used to identify all the trivial fragments which contain the task and their positions within the RPST. Subsequently, this information can be employed to identify the smallest SESE fragment which contains the input task. To ensure that we indeed consider all the possibilities when identifying the smallest SESE fragment, we perform a systematic search which is based on the classification of the *RPST edges*, i.e., parent-child relations between canonical fragments of the RPST. The classification stems from the classes of individual fragments. Theorem 3.12 and the fact that triconnected component subgraphs are derived from triconnected components allow inheriting classes of canonical fragments from triconnected components. Hence, a canonical fragment is of a certain class if it corresponds to a triconnected component subgraph which is derived from a triconnected component of the very same class. Therefore, (i) a canonical fragment is *trivial* (T) if it is composed of a single edge, and (ii) a canonical fragment is *polygon* (P), *bond* (B), or *rigid* (R) if it corresponds to a triconnected component subgraph derived from a maximal polygon, a maximal bond, or a rigid split component, respectively.

A single trivial fragment in isolation is of limited interest when identifying an *abstraction candidate* – the smallest SESE fragment which contains the task. A trivial fragment either connects less than two tasks, in which case the abstraction step that substitutes the trivial fragment with a fresh task cannot do any generalization on tasks in the model, or it connects two tasks, in which case the trivial fragment can be analyzed in the context of its parent polygon canonical fragment.

The RPST of a normalized TTG can have the RPST edges of seven classes; if one ignores relations that include trivial canonical fragments. These are (P, B), (P, R), (B, P), (B, R), (R, P), (R, B), and (R, R) classes; where, for instance, (P, R) edge represents the relation between a parent polygon canonical fragment and its child rigid canonical fragment, see relation between fragments $P1$ and $R1$ in Figure 3.12. Note that (P, P) and (B, B) edges cannot occur in the RPST of a normalized TTG; these relations are always recognized as canonical fragments of either polygon or bond class within triconnected component subgraphs derived from maximal polygon or maximal bond split component, respectively.

Out of seven RPST edge classes, four describe relations of polygon canonical fragments: (P, B), (P, R), (B, P), and (R, P). These RPST edges are of a particular interest for the triconnected abstraction technique, as only maximal polygon triconnected components of normalized TTGs can be composed from task nodes, which are proposed to be used to trigger abstraction steps; every task node

is a non-boundary node of some maximal polygon triconnected component. Note that maximal bond and rigid triconnected components of a normalized TTG are composed of nodes that have at least three incident edges in the TTG; we refer to such nodes as gateway nodes. We assume that gateway nodes are used to define the routing logic of the model and, thus, do not encode any observable action which can be classified either as significant or insignificant in the model. Furthermore, note that boundary nodes of triconnected components are always gateway nodes. Next, we propose individual abstraction rules which exploit all possible structural relations of polygon canonical fragments in RPSTs of normalized TTGs.

Trivial Abstraction

A task in a behavioral model can be directly preceded and/or directly succeeded by another task. We implement abstraction of such a task by aggregating it with one of its neighbors. Any maximal sequence of tasks in a behavioral model forms a single maximal polygon triconnected component which is recognized within the RPST as a polygon fragment. Therefore, a *trivial abstraction* is performed locally, i.e., by aggregating a single trivial fragment inside its parent polygon fragment.

Figure 4.1 exemplifies trivial abstraction. The original maximal polygon triconnected component is given on the left of the figure. The pentagon is a maximal sequence of three task nodes: a, b, and c. Nodes y and z are the boundary gateway nodes. Observe that, in the figure, we visualize triconnected components with directed arcs; these are the arcs of the behavioral model, whereas dashed lines represent virtual edges. Task b is suggested as insignificant, highlighted with grey background and written with bold typeface on the left of the figure. If we were to abstract from task a or c, the selection of the neighbor task to aggregate with would be obvious – it would be task b. In the case of task b triggering abstraction, the selection of a neighbor to aggregate with is delegated to the abstraction control mechanism; in the simplest case this choice can be nondeterministic. In the example, task a is selected to be aggregated with task b; the abstraction candidate is enclosed in the region with a dotted borderline and constitutes a single trivial fragment $T1$ (please refer to the figure).

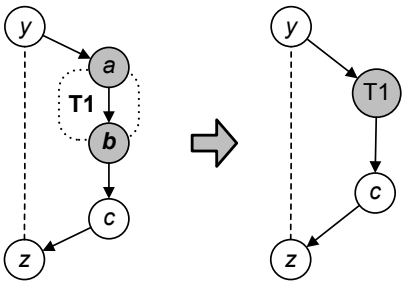


Figure 4.1. Trivial abstraction

The maximal polygon triconnected component on the right of Figure 4.1 is the result of the trivial abstraction step. In the resulting triconnected component, tasks a and b get aggregated into one task $T1$, which semantically corresponds to the task of first performing task a and then accomplishing task b. The triconnected component keeps its structural class – the maximal polygon class. Trivial abstraction is always localized either within the (B, P), or within the (R, P) RPST edge, or is performed within the root polygon canonical fragment.

Polygon Abstraction

A maximal sequence of tasks in a behavioral model can consist of one task. This task can be structured in a sequence with canonical fragments of bond and rigid classes; the relations are reflected by (P, B) and (P, R) RPST edges. If such a task is considered insignificant for the purpose of the model and must be abstracted, one can perform a *polygon abstraction* so that the task gets aggregated with a canonical fragment which directly precedes or directly succeeds the task.

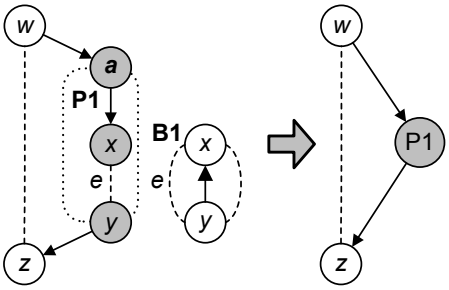


Figure 4.2. Polygon abstraction

Figure 4.2 exemplifies polygon abstraction. The original maximal polygon triconnected component is given on the left of the figure. The pentagon is composed of task a and four gateways: w, x, y, and z. Task a is suggested as insignificant, highlighted with grey background and written with bold typeface on the left of the figure. In the example, task a has no neighbor task; however, it directly precedes maximal bond triconnected component $B1$. Nodes x and y are the boundary nodes of $B1$. The relation between the maximal polygon and maximal bond $B1$ is captured by virtual edge e. Task a is selected to be aggregated with bond $B1$ as it is the only neighbor of task a; again, similar as in the case of trivial abstraction, the selection is delegated to the abstraction control mechanism. The abstraction candidate is enclosed in the region with a dotted borderline and corresponds to a non-maximal polygon split graph $P1$.

The maximal polygon triconnected component on the right of Figure 4.2 is the result of the polygon abstraction step. In the resulting triconnected component, task a and maximal bond $B1$ get aggregated into one task $P1$, which semantically corresponds to the task of first performing task a and then accomplishing the whole fragment $B1$. The triconnected component keeps its structural class – the maximal polygon class. Though we have exemplified polygon abstraction with the help of a task and a maximal bond, it can as well be applied in a straightforward manner for the case of a task and a neighbor rigid split component.

Bond Abstraction

Trivial and polygon abstractions tend to aggregate maximal polygon triconnected components into triangle components. A *triangle* triconnected component is a triangle split component of a behavioral model composed of a single task and two boundary gateway nodes, see the result of the polygon abstraction above with w and z boundary nodes and task $P1$. If the only task of a triangle component is considered insignificant for the purpose of the model, it can be aggregated with (a part of) its parent triconnected component (canonical fragment). If the parent triconnected component is of class bond, we speak about a *bond abstraction*. The task gets aggregated with some child triconnected component of the parent

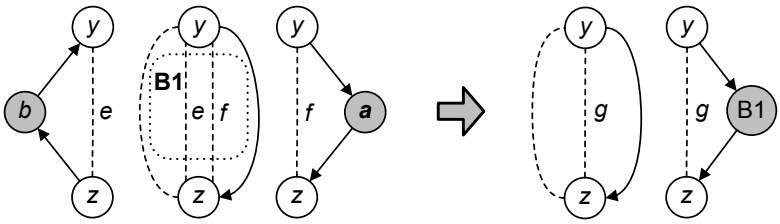


Figure 4.3. Bond abstraction

triconnected component. The selection of a child component to aggregate with is delegated to an abstraction control mechanism.

Figure 4.3 exemplifies bond abstraction. On the left of the figure one can see three triconnected components: a maximal bond and two maximal polygons. Note that both maximal polygons are triangle components. Task a is suggested as insignificant, highlighted with grey background and written with bold typeface on the left of the figure. The task is part of a triangle component whose parent is the maximal bond triconnected component (both share virtual edge f). In the example, task a is selected to be aggregated with the child triconnected component of the maximal bond that contains virtual edge e; the abstraction candidate is enclosed in the region with a dotted borderline and corresponds to a non-maximal bond split graph $B1$.

On the right of Figure 4.3, one can see the result of the bond abstraction. Two triangle components on the left of the figure get aggregated into one triangle component with task $B1$ on the right of the figure. Task $B1$ semantically corresponds to the task of iteratively performing tasks a and b. The resulting triangle component is a child of the maximal bond; this relation is described by virtual edge g. Note that a maximal bond triconnected component may eventually evolve into a triangle component by performing a series of bond abstractions, e.g., if one decides to abstract task $B1$ on the right of Figure 4.3, then the only option is to aggregate $B1$ with trivial fragment (y, z) of the parent bond component.

Rigid Abstraction

In the situation when the parent component of a triangle component is of the rigid class, and the only task of the triangle component is considered insignificant for the purpose of the model, we speak about a *rigid abstraction*. During a rigid abstraction, the insignificant task gets aggregated with the whole parent rigid component and, hence, the abstraction is performed within an (R, P) RPST edge.

Figure 4.4 exemplifies rigid abstraction. On the left of the figure one can see a rigid component and its child triangle component; the relation is captured with the help of virtual edge e. The boundary nodes w and z of the rigid component are highlighted with a thick borderline. Task a is suggested as insignificant in the model, highlighted with grey background and written with bold typeface on the left of the figure. In the example, task a is suggested for aggregation with the

parent rigid component; the abstraction candidate is enclosed in the region with a dotted borderline and corresponds to the whole rigid component $R1$.

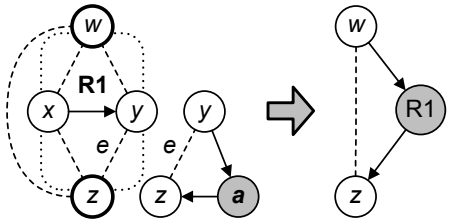


Figure 4.4. Rigid abstraction

The triangle component on the right of Figure 4.4 is the result of the rigid abstraction step. In the resulting triangle component, task a and rigid $R1$ get aggregated into one fresh task $R1$, which semantically corresponds to the task of performing the whole rigid component. The directed arcs in the resulting triangle component hint at the fact that w is the entry and z is the exit of the rigid component on the left of Figure 4.4, i.e., in the abstract model we enter execution of the fresh task from the entry of the canonical fragment which induced the fresh task.

4.2.2. Abstraction Algorithm

Section 4.2.1 proposed four abstraction rules. The rules cover all possible structural relations of a task in a behavioral model. In this section, we organize individual abstraction rules into a procedure which handles a single abstraction step of a task in a behavioral model. As input, the algorithm obtains a TTG (a behavioral model) and one of its task nodes to abstract, i.e., a node with at most one direct predecessor and at most one direct successor. As the result of applying the algorithm, the task node gets abstracted in the TTG; the smallest SESE fragment which contains the task gets substituted with a fresh abstract task of a higher abstraction level. Given all of the above, Algorithm 2 formalizes the principles of the triconnected abstraction.

Algorithm 2: The triconnected abstraction step

Input: A normalized TTG $G = (V, E, \ell)$
Input: A task $v \in V$ of G

```
1  if G is composed of a single node v and no edges then return
2  Compute T — the TTC (Definition 3.6) of G // T defines the RPST
       of G
3  Get p ∈ T – the maximal polygon component in T that contains v
4  if p is triangle then
5  |  Get c ∈ T – the parent component of p in T
6  |  if c is bond then Perform bond abstraction of v
7  |  if c is rigid then Perform rigid abstraction of v
8  else
9  |  if v has the neighbor task in p then
10 |  |  Perform trivial abstraction of v
11 |  else
12 |  |  Perform polygon abstraction of v
```

Algorithm 2 orchestrates individual abstraction rules and attempts to aggregate a minimal number of tasks in a single abstraction step. The algorithm starts by performing a simple check: If the input TTG is composed of a single vertex, there is nothing to abstract and the algorithm terminates (line 1). Otherwise, the algorithm proceeds by computing the TTC of the given graph (line 2). Note that the tree of the triconnected components of a normalized TTG defines its RPST, see Section 3.3.1. At line 3, the algorithm discovers the maximal polygon component p in the TTC that contains the task which triggered abstraction. There is always exactly one such maximal polygon component. The maximal polygon is then used to identify which abstraction rule must be applied. If the maximal polygon is triangle, the choice of abstraction rule to apply is carried out based on the parent component of p. The parent component is identified at line 5 of the algorithm. If the parent component is of class bond, then the bond abstraction is applied (line 6). If the parent component is of class rigid, then the rigid abstraction is applied (line 7). If p is not triangle, the algorithm suggests to apply either the trivial (line 10), or the polygon abstraction (line 12). The choice of the rule is based on the neighbors of the task which triggered the abstraction.

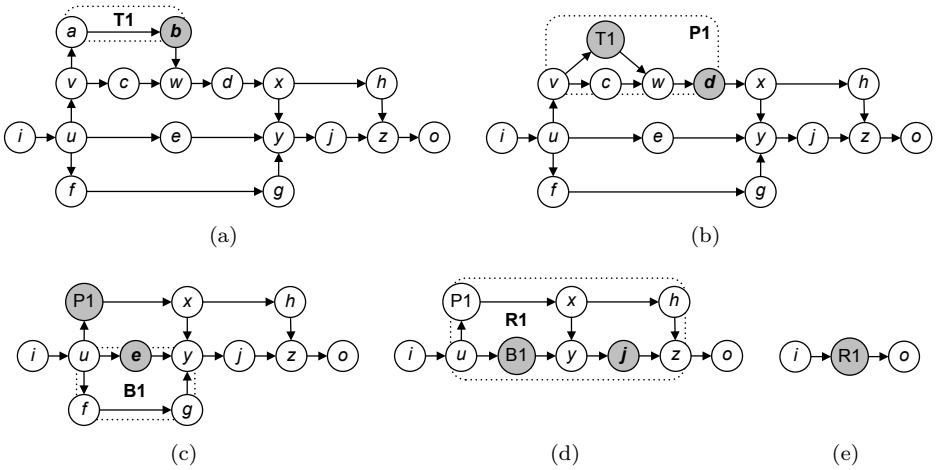


Figure 4.5. The triconnected abstraction

Algorithm 2 provides the formal relation between an original model and its abstract version. Figure 4.5 exemplifies a sequence of the triconnected abstraction steps. Figure 4.5(a) shows the original TTG (it can be used to define the structure of a behavioral model). In the TTG, task b is considered insignificant, which triggers the trivial abstraction of $T1$ (line 10, Algorithm 2). The result of the trivial abstraction is proposed in Figure 4.5(b). Next, task d is considered insignificant, which triggers the polygon abstraction of $P1$ (line 12); the result is proposed in Figure 4.5(c). Afterwards, task e triggers the bond abstraction of $B1$ (line 6), which results in the TTG shown in Figure 4.5(d). Finally, task j triggers the rigid abstraction of $R1$ (line 7), which results in the TTG in Figure 4.5(e).

Different orders of tasks that are considered insignificant in a TTG and trigger abstraction steps lead to different abstract behavioral models. At the end of the triconnected abstraction, the changes in the TTG must be propagated to the corresponding behavioral model. The triconnected abstraction of a task in a TTG can be accomplished in the time linear to the size of the TTG; Algorithm 2 computes the tree of the triconnected components of the input graph and performs local transformations of this tree.

4.3. The Slider-driven Abstraction

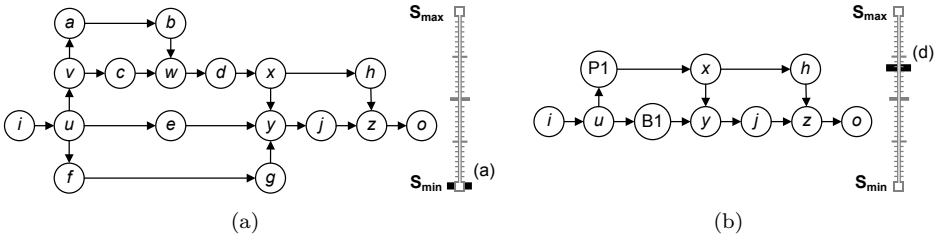
This section answers the *what* question of the methodology for abstraction of behavioral models. In the following, we propose a *slider metaphor* [110] – a tool for enabling gradual control over the abstraction level of behavioral models. We explain how the slider can be employed for distinguishing significant model elements from insignificant ones.

In order to implement abstraction control mechanisms, we propose to distinguish between significant and insignificant model elements based on the annotated properties of these elements. The idea is that annotated properties can be used to enable elements comparison and hence can be used when identifying information relevant for the purpose of the model. We refer to properties that can enable quantitative comparisons of model elements as *abstraction criteria*. For instance, examples of abstraction criteria are [110, 112]: (i) (average) time required to accomplish a task, (ii) (average) cost required to accomplish a task, (iii) average number of occurrences of a task in a single execution of the behavioral model, etc.

Abstraction criteria have quantitative measurement and, thus, criterion values are in a partial order relation. Correspondingly, a partial order relation on criterion values can be transferred to model elements by arranging them according to values of the criterion. For example, if a criterion is the average *time required to accomplish* a task, then a two-minute-task precedes a four-minute-task. The partial order relation enables element classification. It is possible to split model elements into two classes: those with criterion values smaller than and those with criterion values greater than some designed separation point. Elements which are members of the first class are assumed to be insignificant and have to be omitted in the abstract model. Members of the other class are significant and should be preserved in the abstract model. We refer to the separation point according to which the element classes are constructed as the *abstraction threshold*. Assuming an abstraction threshold of three minutes in the example discussed above, the two-minute-task is insignificant and has to be reduced. On the other hand, the four-minute-task is significant and should be preserved in the abstract model.

An *abstraction slider* is an object that operates on a slider interval $[S_{min}, S_{max}]$. The interval is constrained by the minimum (S_{min}) and maximum (S_{max}) values of the abstraction criterion. The slider specifies the criterion value as a slider state $s \in [S_{min}, S_{max}]$ and allows the operation of a state change within this interval.

An abstraction slider regulates the amount of elements preserved in an abstract model; the slider state can be used to specify an abstraction threshold value. In the simplest case a user specifies an arbitrary value used as a threshold, which means



(a) (b)

Figure 4.6. Abstraction slider

that the slider interval is $[-\infty, +\infty]$. The challenge for a user in this approach is to inspect a model in order to choose a meaningful threshold value. A threshold value which is too low leads to all model elements being treated as significant, i.e., no nodes or edges are reduced. On the other hand, a threshold which is too high may result in all elements being treated as insignificant. To avoid such confusing situations, the user should be supported by suggesting an interval in which all the "useful" values of abstraction criterion lie. The support is specific for each abstraction criterion. For instance, if the abstraction criterion is the average time required to accomplish a task, then it is reasonable to configure a slider so that $S_{min} = 0$ and S_{max} is set to the average time required to execute the model. Of course, such an approach must be supported by techniques capable of computing average time required to execute the model (part of the model), e.g., [26]. If the above described approach is taken, then the minimum threshold value suggests no elements as insignificant, the maximum suggests all as insignificant, and every other threshold value suggests a share of all elements as insignificant.

Once the threshold value is specified, insignificant elements must be reduced, e.g., by applying the technique from Section 4.2. The elements must be reduced one by one, starting from the one for which the criterion value is the lowest. Once all insignificant elements are reduced, one obtains the resulting abstract model. For instance, coming back to our example with average time, the purpose of the abstraction can be to compute a model which contains no tasks that require less than a few seconds to complete and to group them into abstract tasks.

Figure 4.6 shows two TTGs from Figure 4.5. The TTG in Figure 4.6(a) corresponds to a detailed behavioral model, i.e., no task nodes are reduced. This situation corresponds to the slider state set to its minimum value, see the figure. If a user wants to compute an abstract model, (s)he must change the state of the slider, see in Figure 4.6(b). The new slider state suggests that the user does not want to see task nodes below the threshold. The state change operation triggers a sequence of abstraction steps, e.g., the ones shown in Figures 4.5(a), 4.5(b), and 4.5(c), and leads to the abstract model in Figure 4.6(b).

4.4. Fragment Extracts

In this section, we propose a technique which, given a behavioral model and one of its canonical fragments, computes the *extract of the fragment* from the

behavioral model. The idea of the technique is the following: Given a TTG, which defines a behavioral model, we extract the given canonical fragment from the context of the TTG and abstract from all its child canonical fragments except trivial fragments and triangle fragments, where a triangle fragment is a fragment of the TTG derived from its triangle triconnected component. The extract is then another TTG (behavioral model) which captures the high-level behavior of one specific canonical fragment. We envision that the technique can find its use when modularizing problems which deal with the analysis of behavioral models. Specifically, we shall employ this technique in Part III to modularize the structuring problem.

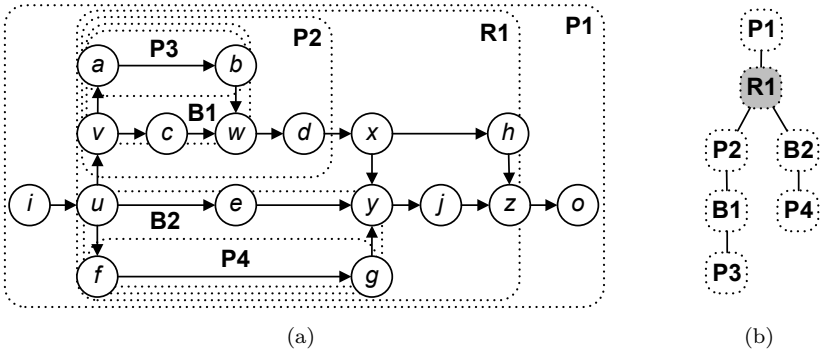


Figure 4.7. (a) A TTG and its canonical fragments, (b) the (simplified) RPST of (a)

Figure 4.7(a) shows the TTG from Figure 4.5(a) and some of its canonical fragments. Note that trivial and triangle fragments are not visualized in the figure. Figure 4.7(b) shows the simplified RPST of the TTG; again, trivial and triangle fragments are omitted. Since the extraction technique is straightforward to understand, we omit its formal definition and specify by means of an example.

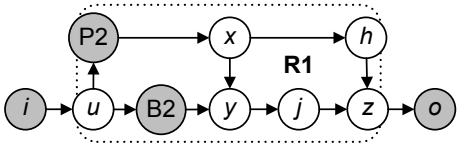


Figure 4.8. Extract of fragment $R1$ from the TTG in Figure 4.7(a)

Figure 4.8 shows the extract of fragment $R1$ (highlighted with grey background in Figure 4.7(b)) from the TTG in Figure 4.7(a). As shown in Figure 4.7, $R1$ has two non-trivial and non-triangle child fragments: $P2$ and $B2$. In the extract, these fragments get abstracted into tasks $P2$ and $B2$, respectively (highlighted with grey background in Figure 4.8). Nodes h, j, u, x, y, and z are the nodes of the original TTG. Note that nevertheless nodes i and o are present in both TTGs, see in Figure 4.7(a) and in Figure 4.8, those in Figure 4.8 are the fresh ones; they are designed to explicitly mark the entry and the exit of the extracted fragment (also highlighted with grey background in the figure). The explicit source and/or

explicit sink node may be omitted in the extract if one can uniquely identify the source and/or sink node of the extract without introducing fresh node(s). Observe that this is the case in Figure 4.8; when without i and o, nodes u and z can still be recognized as the source and the sink node of the extract, respectively.

The extract of a canonical fragment captures its high-level behavior by abstracting from low-level details of its child fragments and taking the fragment out of the context of the whole behavioral model. Due to the fact that canonical fragments can be discovered in the time linear to the size of the TTG, refer to Section 3.3 and Section 3.4, the extract of a canonical fragment can also be computed in the time linear to the size of the TTG.

4.5. Conclusion

In this chapter, we proposed an abstraction methodology for behavioral models – an approach to derive behavioral models of high abstraction levels from the detailed ones. We argued that the abstraction task can be decomposed into two independent subtasks: learning process model elements which are insignificant (abstraction *what*) and abstracting from those elements (abstraction *how*).

Furthermore, in this chapter, we proposed one concrete instantiation of the abstraction methodology. The *how* question of the methodology is answered by means of the triconnected abstraction technique. The triconnected abstraction technique defines structural model transformations and can be generalized to any process modeling notation which uses directed graphs as the underlying formalism. The technique can be trivially generalized to become applicable without any structural limitations, i.e., for an arbitrary MTG. The generalization step is similar to the step of generalizing the technique for RPST computation of normalized TTGs, to the technique for the RPST computation of general TTGs and MTGs, see Section 3.3.2 and Section 3.4. The *what* question of the methodology is answered by means of the abstraction slider – an abstraction control mechanism. An abstraction slider can aid when telling significant model elements from insignificant ones. The slider state specifies the abstraction threshold and, thus, all the insignificant elements of the behavioral model, i.e., elements for which the value for the abstraction criterion is below the threshold. Next, insignificant elements get reduced with the help of the triconnected abstraction technique.

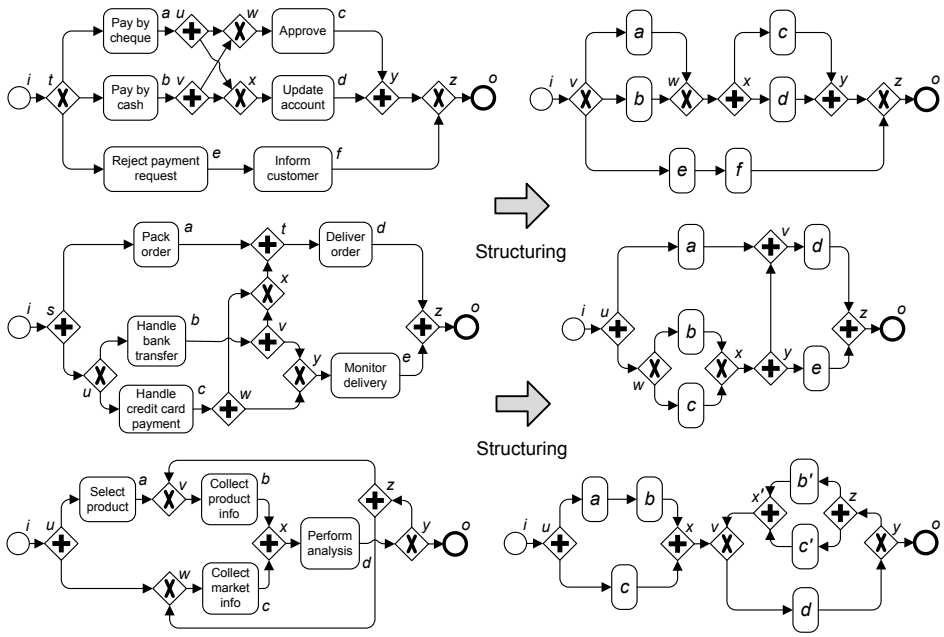
Finally, in this chapter, we proposed a technique for extracting the core behavior captured in a SESE fragment from the overall context of the behavioral model. This technique can be useful when modularizing problems that deal with the analysis of behavioral models. In particular, in the next part of this thesis, we shall employ this technique for the modularization of the structuring problem.

In future works, we envision other instantiations of the abstraction methodology for behavioral models, i.e., those which employ abstraction rules other than ones proposed by the triconnected abstraction technique and which use abstraction control mechanisms other than the slider-driven one. For more results on the abstraction methodology for behavioral models please refer to our originative works on business process model abstraction (BPMA) in the context of business process management [110, 111, 112].

Part III.

Structuring

5. Structuring Foundations



This chapter serves two purposes: (i) it formally defines all the previously intuitively introduced notions on structuring, and (ii) it provides the still missing (structuring related) preliminaries. Section 5.1 uses the notion of process components to define the well-structured property of process models. Section 5.2 is devoted to the motivation and the formal discussion of fully concurrent bisimulation – a behavioral equivalence notion adopted by all the subsequently proposed structuring techniques. Section 5.3 discusses existing structuring techniques which can deliver fully concurrent bisimilar structured process models. Section 5.4 proposes a convenient way for deciding the fully concurrent bisimulation of two net systems of a special kind. Finally, Section 5.5 presents the preliminary notions on the technique of the net system unfolding.

5.1. Well-structuredness and Process Components

This section formally defines the well-structured property of process models, see Section 5.1.1. The definition is based on the notion of a *process component*, which can be seen as a special type of a single-entry-single-exit component. Section 5.1.2 proposes a taxonomy of process components.

5.1.1. Well-structured Process Models

In Section 1.2, we have intuitively defined the well-structured property of behavioral models. In this section, we give a formal form to that intuition using process models as a concrete example of behavioral models. We propose to structurally classify a process model based on the properties of its parse tree, in particular the RPST (Chapter 3). This proposal is due to the fact that the notion of the RPST fragment coincides with the notion of the SESE component used in the intuitive definition. Hence, the RPST is a natural candidate for the characterization of structuredness of behavioral models.

A SESE component of a process model is defined by a fragment (Definition 3.3) of the graph that is employed in formalizing the process model.

Definition 5.1 (Process component).
Let $PM = (A, G, C, type, \mathcal{A}, \mu)$ be a process model. A canonical fragment $F \subseteq C$ of the MTG $(A \cup G, C, \ell)$, where ℓ is the identity function on C, is called a *process component*, or a *component*, of PM. ⌟

Process components are defined by canonical fragments. Note that we speak of interior, boundary, entry, and exit nodes of a process component based on their classification in the subgraph formed by the canonical fragment that defines the process component (Definition 3.2). Furthermore, we inherit the classification of process components from the classification of the triconnected components which are used to derive canonical fragments.

Definition 5.2 (Trivial, Polygon, Bond, Rigid component).
Let F be a process component of a process model PM.

- F is a *trivial* component, iff F is singleton.
- F is a *polygon* component, iff there exists a sequence $(r_0, \ldots, r_n)$, $n \in \mathbb{N}$, where $\mathbb{N}$ is the set of natural numbers excluding zero, of components of PM, s.t. $F = \bigcup_{i=0}^{i=n} r_i$, the entry of F is the entry of r_0, the exit of F is the exit of r_n, and the exit of r_j is the entry of r_{j+1}, $0 \leq j < n$.
- F is a *bond* component, iff there exists a set R of components of PM, s.t. $F = \bigcup_{r \in R} r$ and every component in R has the same boundary nodes as F.
- F is a *rigid* component, iff F is neither a trivial, nor a polygon, nor a bond component. ⌟

The class of a process component can be trivially determined based on the class of the triconnected component which is used to derive the process component. Figure 5.1 exemplifies process components of two process models from Figure 1.1. Every box with a dotted border defines a component which is composed of the control

flow arcs that are inside or intersect the box. Observe that Figure 5.1 does not show all process components. In the following, we shall often ignore the visualization of simple components, where a *simple* component is either a trivial component, or a polygon component composed of two trivial components. Therefore, we do not explicitly show polygon process components $(\{(u,a)\},\{(a,v)\})$, $(\{(u,b)\},\{(b,w)\})$, $(\{(x,c)\},\{(c,z)\})$, and $(\{(y,d)\},\{(d,z)\})$ in Figure 5.1(a), and polygon process components $(\{(w,a)\},\{(a,x)\})$, $(\{(w,b)\},\{(b,x)\})$, $(\{(y,c)\},\{(c,z)\})$, and $(\{(y,d)\},\{(d,z)\})$ in Figure 5.1(b). Note that again, the names of components hint at their class, i.e., $P1$ is a polygon, $B1$ is a bond, and $R1$ is a rigid component.

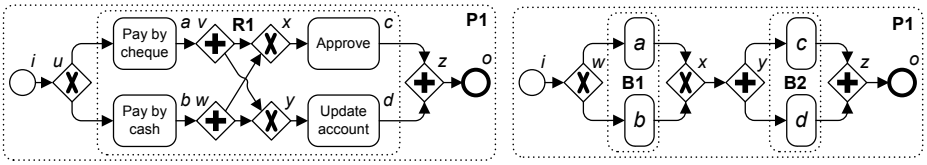


Figure 5.1. Process models: (a) unstructured, and (b) well-structured

The set of all process components of a process model defines the set of all its SESE components. A process component is a SESE component. Moreover, recall from Chapter 3 that canonical fragments of a graph define all its fragments (also the non-canonical ones), where non-canonical fragments correspond to subsequences of maximal polygons and subsets of maximal bonds. Therefore, the set of all process components of a process model determines its structuredness.

Definition 5.3 (Well-structured process model).
A process model PM is *(well-)structured*, iff PM contains no rigid process component; otherwise the process model is *unstructured*.

By definition, we postulate that a process model is well-structured if and only if its RPST is composed only of trivial, polygon, and bond components. In a process model that satisfies this condition, every split has a corresponding join, and vice versa, such that the part of the model between the split and the join is a SESE component (by virtue of the definition of a bond). If one would be able to discover a pair of a corresponding split and a join inside a rigid process component, then this pair would be the boundary pair of a bond component.

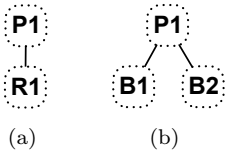


Figure 5.2. The (simplified) RPSTs of the process models in Figure 5.1

Well-structuredness of a process model can be determined by computing its RPST and checking whether any of its canonical fragments are derived from a rigid triconnected component. Figure 5.2 shows the RPSTs of the process models in Figure 5.1 as tree-like structures. Again, we ignore simple components. The process model in Figure 5.1(a) contains rigid component $R1$ and is, therefore, unstructured. In contrast, the process model in Figure 5.1(b) contains no rigid components and is well-structured.

5.1.2. Taxonomy of Process Components

A process model is unstructured if it contains a rigid component. Therefore, if one could transform every rigid component into an equivalent well-structured component (or a set of well-structured components), the entire process model could be structured by traversing its RPST bottom-up and replacing encountered rigid components by equivalent well-structured components.

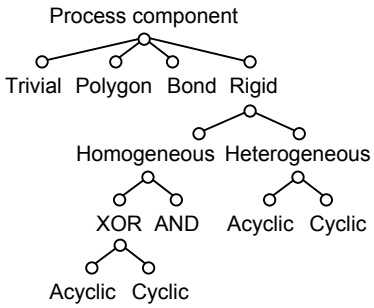


Figure 5.3. Taxonomy of components

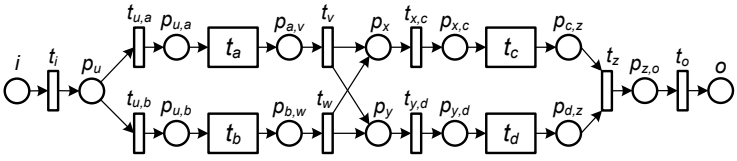
The existing methods for structuring rigid components differ depending on the types of gateways present in a rigid component and whether the rigid component contains cycles or not. More precisely, above we talk about gateways of process component extracts, refer to Section 4.4, i.e., child components of a rigid process component must be abstracted to achieve the modularization of the structuring problem. Accordingly, we found it useful to further classify (extracts of) rigid components as follows: A homogeneous rigid contains either only *xor* or only *and* gateways. We call these rigids *(homogeneous) xor rigids* and *(homogeneous) and rigids*, respectively. A heterogeneous rigid contains a mixture of *and*/*xor* gateways. Heterogeneous and homogeneous *xor* rigids are further classified into cyclic, if they contain at least one cyclic path, or acyclic. Importantly, we do not classify homogeneous *and* rigids as cyclic or acyclic, as process models with cyclic *and* rigids are unsound [147]. Based on this background, a taxonomy of process components is provided in Figure 5.3.

5.2. Behavioral Equivalence of Process Models

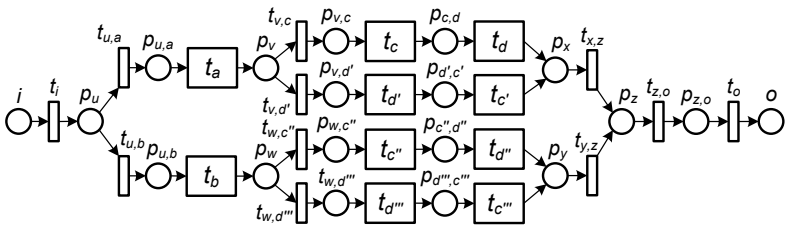
An unstructured process model and its well-structured version are structurally different, but behaviorally equivalent. This immediately raises the question of what notion of behavioral equivalence is the most applicable in the context of the process model structuring problem. There exist various notions of behavioral equivalence for concurrent systems [139]. This section motivates the *fully concurrent bisimulation* [11] as the equivalence notion appropriate for structuring process models, as it sufficiently preserves the level of concurrency of observable operations in equivalent systems.

A common notion of behavioral equivalence for concurrent systems is that of *bisimulation*. Bisimulation has been introduced in [101] as a concept that is equivalent to observational equivalence [88]. Since then, the notion of bisimulation has gained considerable attention in the literature. Related equivalence notions are those of *weak bisimulation* and *branching bisimulation*. These notions have been advocated as being suitable for comparing process models [68]. Weak bisimulation abstracts from silent transitions, i.e., silent transitions may be executed but their execution is not visible for an external observer, cf., [89]. As pointed out in [140],

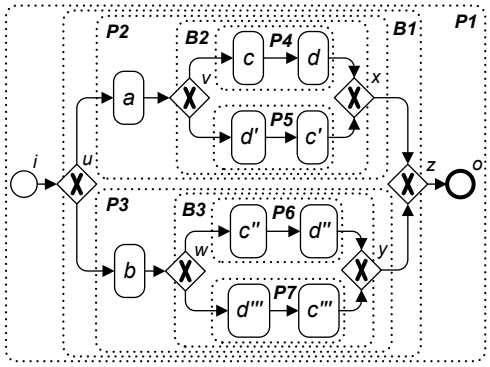
weak bisimulation does not preserve branching time for silent transitions. This observation led to the introduction of the notion of branching bisimulation.



(a)



(b)



(c)

Figure 5.4. (a) A WF-net that corresponds to the process model in Figure 5.1(a), (b) sequential simulation of (a), and (c) the process model that net in (b) corresponds to

Usually, bisimulation is defined in terms of execution sequences. For instance, all of the above mentioned notions of bisimulations adopt an interleaving semantics, i.e., no two tasks are executed exactly at the same time. Thus, a concurrent system and its sequential simulation are considered equivalent. For example, Figure 5.4(b) shows the sequential simulation of the WF-net in Figure 5.4(a), which is the net that corresponds to the unstructured process model in Figure 5.1(a). The nets in Figure 5.4(a) and Figure 5.4(b) are weakly bisimilar. Accordingly, we speak of a weak bisimulation of the process models that these nets correspond to, see Figure 5.1(a) and Figure 5.4(c). The process model in Figure 5.4(c) is also well-structured; note that its RPST contains no rigids. However, the process model contains no parallel branch and, thus, does not capture the fact that tasks

c and d can be enabled and executed concurrently. Instead, the concurrency is modeled by means of interleaving the occurrences of the corresponding tasks.

One can trivially construct a well-structured process model, like the one in Figure 5.4(c), by computing all *sequential runs* of a given unstructured process model and afterwards performing sequential merging of individual runs by means of *xor* gateways. The amount of task duplicates in this case greatly depends on the quality of the merging of individual runs. Figure 5.4(c) shows the result of applying one possible merging strategy. Yet, one can envision a merging strategy where activities a and b are duplicated, or a strategy where activities c and d are duplicated only once. The proposed structuring method is complete, but if we start with an unstructured process model containing *and* gateways, we always obtain a (much larger) structured process model without any parallel branches; these are the general tendencies of the described approach, which can be already recognized in Figure 5.4(c).

Accordingly, we adopt a notion of bisimulation that preserves the level of concurrency of observable operations, viz. *fully concurrent bisimulation* (FC-bisimulation) [11]. FC-bisimulation is a Petri net analog of the behavioral structure bisimulation [116] or the history preserving bisimulation on event structures [141]. FC-bisimulation is defined in terms of *concurrent runs* of a system, a.k.a. *processes* in the literature (but not to be confused with "business processes" or workflows). Every process of a system can be expressed as another net with a particular structure – a *causal* net.

Definition 5.4 (Causal net).
A net $N = (B, E, G)$ is a *causal* net, iff :

- for each $b \in B$ holds $|\bullet b| \leq 1$ and $|b \bullet| \leq 1$, and
- N is acyclic, i.e., G^+ is irreflexive.

Elements of E are called *events* and elements of B are called *conditions*. We also introduce *ordering relations* [96], which will be used in the subsequent sections as an instrument for reasoning about the behavior of nets.

Definition 5.5 (Ordering relations).
Let $N = (P, T, F)$ be a net and let $x, y \in P \cup T$ be its nodes.

- x and y are in *causal* relation, written $x \rightsquigarrow_N y$, iff $(x, y) \in F^+$. y and x are in *inverse causal* relation, written $y \leftsquigarrow_N x$, iff $x \rightsquigarrow_N y$.
- x and y are in *conflict*, $x \#_N y$, iff there exist distinct transitions $t_1, t_2 \in T$, s.t. $\bullet t_1 \cap \bullet t_2 \neq \varnothing$, and $(t_1, x), (t_2, y) \in F^*$. If $x \#_N x$, then x is in *self-conflict*.
- x and y are *concurrent*, $x \|_N y$, iff neither $x \rightsquigarrow_N y$, nor $y \rightsquigarrow_N x$, nor $x \#_N y$.

The set $\mathcal{R}_N = \{\rightsquigarrow_N, \leftsquigarrow_N, \#_N, \|_N\}$ forms the *ordering relations* of N.

The *observable* ordering relations of a net are formed by its ordering relations where every relation is restricted to the set of observable transitions of the net.

Definition 5.6 (Observable ordering relations).
Let $N = (P, T, F, \mathcal{T}, \lambda)$ be a labeled net and let $T' \subseteq T$ be its observable transitions. The set of ordering relations of N where each relation is restricted to the set T' forms the *observable ordering relations*, or λ-*ordering relations*, $\mathcal{R}_\lambda$ of N, i.e., $\mathcal{R}_\lambda = \{\rightsquigarrow_N \cap (T' \times T'), \leftsquigarrow_N \cap (T' \times T'), \#_N \cap (T' \times T'), \|_N \cap (T' \times T')\}$.

It is easy to see that any two nodes in a causal net are either in (inverse) causal relation or concurrent. In the following we omit the subscripts of ordering relations where the context is clear. In order to define a process, we lack the notion of a *cut*. A *co-set* is a set of pairwise concurrent places. A maximal co-set with respect to inclusion is a *cut*. Then, a process is defined as follows.

Definition 5.7 (Process).
A *process* $\pi = (N_\pi, \rho)$ of a system $S = (N, M_0)$, $N = (P, T, F)$, consists of a causal net $N_\pi = (B, E, G)$ and a function $\rho : B \cup E \to P \cup T$:

- $\rho(B) \subseteq P,\ \rho(E) \subseteq T$,
- $Min(N_\pi)$ is a cut, which corresponds to the initial marking M_0, that is $\forall\ p \in P : M_0(p) = |\rho^{-1}(p) \cap Min(N_\pi)|$, and
- $\forall\ e \in E\ \forall\ p \in P : (F(p, \rho(e)) = |\rho^{-1}(p) \cap {\bullet e}|) \wedge (F(\rho(e), p) = |\rho^{-1}(p) \cap e \bullet|)$.

A process π of S is *initial*, iff $E = \varnothing$. ⌟

A process π' *is an extension of* a process π if it is possible to observe π before one observes π'. Consequently, process π *is a prefix of* π'.

Definition 5.8 (Prefix, Process extension).
Let $\pi = (N_\pi, \rho)$, $N_\pi = (B, E, G)$, be a process of a net system. Let c be a cut of N_π and let $c^\downarrow$ be the set $\{x \in B \cup E \mid \exists\ y \in c : (x, y) \in G^*\}$. A process π_c *is a prefix of* π up to (and including) c, iff $\pi_c = ((B \cap c^\downarrow, E \cap c^\downarrow, F \cap (c^\downarrow \times c^\downarrow)), \rho|_{c^\downarrow})$. A process π' *is an extension of* process π, if π is a prefix of π'. ⌟

In order to define fully concurrent bisimulation, we need two auxiliary definitions: λ-abstraction of a process, which is a process footprint that ignores silent transitions, and the order-isomorphism of λ-abstractions.

Definition 5.9 (Abstraction of a process of a labeled system).
Let $S = (N, M_0)$, $N = (P, T, F, \mathcal{T}, \lambda)$, be a labeled system and let $\pi = (N_\pi, \rho)$, $N_\pi = (B, E, G)$, be a process of S. The λ-abstraction of π, denoted by $\alpha_\lambda(\pi) = (E_\pi, \prec, \lambda_\pi)$, is defined by the set of observable events $E_\pi \subseteq E$ of N_π, the observable causal relation $\prec$ of N_π , and $\lambda_\pi : E_\pi \to \mathcal{T}$, such that $\lambda_\pi(e) = \lambda(\rho(e))$, $e \in E_\pi$. ⌟

Two λ-abstractions are *order-isomorphic*, if there exists a one-to-one correspondence between events of both abstractions which also preserves the ordering relations of the corresponding events in the abstractions.

Definition 5.10 (Order-isomorphism of abstractions).
Let $\alpha_{\lambda_1} = (E_1, \prec_1, \lambda_1)$ and $\alpha_{\lambda_2} = (E_2, \prec_2, \lambda_2)$ be two λ-abstractions, both with labels in $\mathcal{T}$. Then α_{λ_1} and α_{λ_2} are *order-isomorphic*, denoted by $\alpha_{\lambda_1} \cong \alpha_{\lambda_2}$, iff there is a bijection $\beta : E_1 \to E_2$, such that $\forall\ e \in E_1 : \lambda_1(e) = \lambda_2(\beta(e))$ and $\forall\ e_1, e_2 \in E_1 : e_1 \prec_1 e_2 \Leftrightarrow \beta(e_1) \prec_2 \beta(e_2)$. ⌟

Given all of the above, fully concurrent bisimulation is defined as follows.

Definition 5.11 (Fully concurrent bisimulation).
Let $S_1 = (N_1, M_1)$ and $S_2 = (N_2, M_2)$ be labeled systems, $N_1 = (P_1, T_1, F_1, \mathcal{T}_1, \lambda_1)$ and $N_2 = (P_2, T_2, F_2, \mathcal{T}_2, \lambda_2)$. S_1 and S_2 are *fully concurrent bisimilar*, or *FCB-equivalent*, denoted by $S_1 \approx S_2$, iff there is a set $\mathcal{B} \subseteq \{(\pi_1, \pi_2, \beta)\}$, such that:

(i) π_1 is a process of S_1, π_2 is a process of S_2, and β is a relation between the non-τ events of π_1 and π_2.
(ii) If π_0^1 and π_0^2 are the initial processes of S_1 and S_2, respectively, then $(\pi_0^1, \pi_0^2, \varnothing) \in \mathcal{B}$.
(iii) If $(\pi_1, \pi_2, \beta) \in \mathcal{B}$, then β is an order-isomorphism between the λ_1-abstraction of π_1 and the λ_2-abstraction of π_2.
(iv) $\forall (\pi_1, \pi_2, \beta) \in \mathcal{B}$:
  (a) If π_1' is an extension of π_1, then $\exists (\pi_1', \pi_2', \beta') \in \mathcal{B}$ where π_2' is an extension of π_2 and $\beta \subseteq \beta'$.
  (b) Vice versa. ⌟

Fully concurrent bisimulation defines an equivalence relation on labeled systems that is stricter than weak bisimulation and related notions. The nets in Figure 5.4(a) and Figure 5.4(b) are not fully concurrent bisimilar. Meanwhile, the two nets in Figure 5.4(a) and Figure 5.5 are FCB-equivalent. The net in Figure 5.5 corresponds to the process model in Figure 5.1(b) and, thus, the model in Figure 5.1(b) is the FCB-equivalent well-structured version of the process models in Figure 5.1(a) (with the understanding that two process models are FCB-equivalent, iff the corresponding Petri nets are FCB-equivalent).

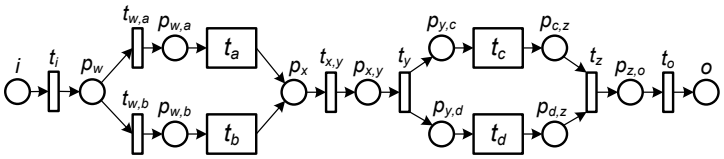


Figure 5.5. A WF-net that corresponds to the process model in Figure 5.1(b)

5.3. Related Work

So far, we have instantiated the structuring problem (refer to Section 1.3) with the notions of the process model (as a behavioral model, see Section 2.5) and the fully concurrent bisimulation (as a behavioral equivalence relation, see Section 5.2). In this section, we look at this particular instance of the structuring problem from the perspective of related works which deal with structuring of behavioral models.

Our setting of the structuring problem is close to the problem of structuring sequential programs, which has been extensively studied for years. In one of his letters, Edsger W. Dijkstra started a discussion with the provocative title "Go To Statement Considered Harmful" [25]. In the absence of Go To statements, programs are composed of structured flow constructs only, a situation that corresponds to our notion of well-structured process models without concurrency constructs. The main idea communicated in the letter is that in the context of sequential programs, Go To statements should be abolished from all "high-level" programming languages. Since that time, many replies to the letter supported or rejected the statement of Dijkstra, for instance [55, 159, 119, 93], with no side being able to provide sound arguments to disarm one another. The partial resolution of the conflict became possible

due to many works on formal techniques for translating unstructured programs with Go To statements into equivalent structured programs [156, 97, 162]. The main outcome of these endeavors is that any sequential program can be structured with only one control flow pattern, viz. forward jump from a loop, requiring the introduction of fresh control variables in structured programs [97]; other flow patterns can be structured by employing code duplications and structured control flow constructs like `if-then-else` or `while-do`.

The results on structuring sequential programs do not hold for process models which comprise concurrency. One of the earliest studies on the problem of structuring process models is that of Bartek Kiepuszewski et al. [67]. The authors showed that not all acyclic process models can be structured by putting forward a counter-example, which essentially boils down to the one in Figure 5.6 (also known as Z-structure, due to the configuration of causal relations between the tasks). The authors showed that there is no well-structured process model that is equivalent to this one under the fully concurrent bisimulation equivalence notion. They do explore some causes of unstructuredness, but neither give a full characterization of the class of models that can be structured, nor do they define any automated transformation.

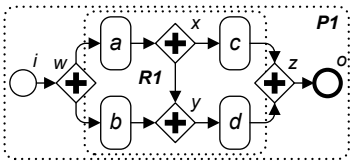


Figure 5.6. Process model (Z-structure)

Some work has been devoted to the characterization of sources of unstructuredness in process models. In [77], the authors present a taxonomy of unstructuredness, that covers acyclic and (partly) cyclic process models. The taxonomy is based on the notion of improper nesting and mismatched pairs. The taxonomy allows analyzing unstructured process models, determine whether they are well-behaved, and whether they can be transformed into equivalent structured models. However, the taxonomy is incomplete, as it does not cover all possible cases of process models that can be structured. Besides, the authors do not define an automated algorithm for structuring unstructured process models.

Other methods simply reuse techniques for structuring sequential programs in order to partly structure process models. [50] proposes a method for structuring sequential parts of a process model based on Go To program transformations, and extends this method to process graphs where concurrent parts are already structured. This method cannot deal with process models which comprise unstructured concurrent threads of control. A similar remark applies to [71], where authors concentrate on structuring of unstructured cyclic flows. In [79], a translation from (unstructured) Petri nets to (structured) BPEL processes is proposed. While the proposed method can handle unstructured concurrent threads of control, it does so by directly expressing them in terms of BPEL's `flow` activity and `links`. Put differently, the method identifies the already structured parts of the process model, but provides no means for structuring the unstructured parts.

In [49], the authors outline a classification of process components (parts of process models) using *region trees*. The authors mention that region trees can be used to structure unstructured process models, however they do not provide

a structuring method, even for acyclic models. In [109], the authors study the influence of "hidden" unstructuredness in process models on their correctness.

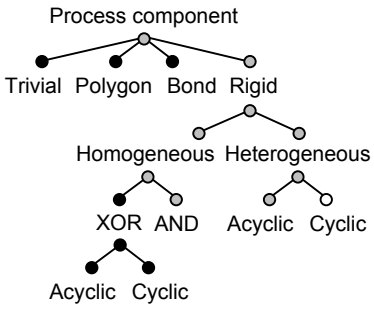


Figure 5.7. Taxonomy of components

Figure 5.7 uses the taxonomy of process components from Figure 5.3 to visualize the coverage of the structuring problem by the existing techniques. Black circles hint at the complete solution. Trivial, polygon, and bond components are structured process components, whereas homogeneous *xor* rigids (both acyclic and cyclic) can be structured by employing techniques on program translation, e.g., [97]. Circles with grey background indicate the existence of partial solutions, which were developed for acyclic models with concurrency. We are not aware of any existing solution which systematically addresses the problem of structuring cyclic process models, even for subclasses of cyclic models, indicated by the empty circle in the figure.

To sum up, to the best of our knowledge, existing techniques approach structuring of process models with concurrency rather superficially. Existing techniques either drop the requirement of preserving concurrency described in the unstructured process model, or the "problematic" parts of the process model are not structured at all and, hence, unstructuredness remains in the resulting process model.

5.4. Behavioral Equivalence and Ordering Relations

The definition of fully concurrent bisimulation from Section 5.2 is abstract and hardly of any use when synthesizing structured nets from unstructured ones. Accordingly, we employ a more convenient way of reasoning about equivalence based on the ordering relations of the special class of nets, viz. *occurrence nets*.

Nets can have forward or backward conflicts, i.e., places with multiple output or input transitions, respectively. This means that a subnet which may cause a transition firing is not unique. An *occurrence net* is a net of a special kind. Occurrence nets forbid backward conflicts and, thus, ensure the unique cause of a transition firing. Essentially, occurrence nets generalize causal nets by allowing forward conflicts. Note that every causal net is also an occurrence net.

Definition 5.12 (Occurrence net).
A net $N = (B, E, G)$ is an *occurrence* net, iff :

- for each $b \in B$ holds $|\bullet b| \leq 1$,
- N is acyclic, i.e., G^+ is irreflexive,
- for each $x \in B \cup E$ the set $\{y \in B \cup E \mid (y, x) \in G^+\}$ is finite, and
- no $e \in E$ is in self-conflict, i.e., $\#_N$ is irreflexive. ⌟

Again, elements of E are called *events* and elements of B are called *conditions*. Note that every two nodes of an occurrence net are either in causal, inverse causal, conflict, or concurrent relation [96].

We say that λ-ordering relations (Definition 5.6) of two labeled occurrence nets are isomorphic, if there exists a mapping between the sets of observable events such that every corresponding pair of events is in the same ordering relation.

Definition 5.13 (Isomorphism of ordering relations).
Let $N_1 = (B_1, E_1, G_1, \mathcal{T}_1, \lambda_1)$ and $N_2 = (B_2, E_2, G_2, \mathcal{T}_2, \lambda_2)$ be two labeled occurrence nets. Let $E_1' \subseteq E_1$ and $E_2' \subseteq E_2$ be observable events of N_1 and N_2, respectively. Two λ-ordering relations $\mathcal{R}_{\lambda_1}$ of N_1 and $\mathcal{R}_{\lambda_2}$ of N_2 are *isomorphic*, denoted by $\mathcal{R}_{\lambda_1} \cong \mathcal{R}_{\lambda_2}$, iff there is a bijection $\gamma : E_1' \to E_2'$, such that:

- $\forall\ e \in E_1' : \lambda_1(e) = \lambda_2(\gamma(e))$, and
- $\forall\ e_1, e_2 \in E_1' : (e_1 \rightsquigarrow_{N_1} e_2 \wedge \gamma(e_1) \rightsquigarrow_{N_2} \gamma(e_2)) \vee (e_1 \reflectbox{$\rightsquigarrow$}_{N_1} e_2 \wedge \gamma(e_1) \reflectbox{$\rightsquigarrow$}_{N_2} \gamma(e_2)) \vee (e_1 \#_{N_1} e_2 \wedge \gamma(e_1) \#_{N_2} \gamma(e_2)) \vee (e_1 \|_{N_1} e_2 \wedge \gamma(e_1) \|_{N_2} \gamma(e_2))$. ⌟

Finally, we show that two occurrence nets with isomorphic ordering relations are FCB-equivalent, and vice versa.

Theorem 5.14. *Let $S_1 = (N_1, M_1)$, $N_1 = (B_1, E_1, G_1, \mathcal{T}_1, \lambda_1)$, and $S_2 = (N_2, M_2)$, $N_2 = (B_2, E_2, G_2, \mathcal{T}_2, \lambda_2)$, be two labeled occurrence systems with natural markings and distinctive labelings. Let $E_1' \subseteq E_1$ and $E_2' \subseteq E_2$ be observable events of N_1 and N_2, respectively, such that there exists a bijection $\psi : E_1' \to E_2'$ for which holds $\lambda_1(e) = \lambda_2(\psi(e))$, for all $e \in E_1'$. Let $\mathcal{R}_{\lambda_1}$ and $\mathcal{R}_{\lambda_2}$ be the λ-ordering relations of N_1 and N_2, respectively. Then, it holds:*

$$S_1 \approx S_2 \Leftrightarrow \mathcal{R}_{\lambda_1} \cong \mathcal{R}_{\lambda_2}.$$

The proof of Theorem 5.14 is in Appendix A.2.

5.5. Unfoldings

An unfolding of a net system is another net that explicitly represents all concurrent runs of the net system in a possibly infinite, tree-like structure [34, 36]. In [84], Kenneth L. McMillan proposed an algorithm for constructing a *finite* initial part of the unfolding, which contains full information about the reachable markings of the net system, viz. a *complete prefix unfolding*. In the following, we present main notions of the theory of unfoldings: Section 5.5.1 presents branching processes – a convenient mechanism for capturing concurrent runs of a net system. Section 5.5.2 presents an unfolding algorithm – the algorithm for constructing maximal branching processes. Finally, Section 5.5.3 presents an algorithm for the construction of the finite initial part of an unfolding which contains information about all reachable markings of the net system. Both algorithms were originally proposed in [37, 38].

5.5.1. Branching Processes

This section discusses branching processes – a partial-order semantics of Petri nets. Every net system can be "unfolded" into an occurrence net. The unfolding procedure allows one to preserve correspondences between nodes of the resulting occurrence net and nodes of the net system. The occurrence net together with a

mapping of its nodes to the nodes of the net system is called a *branching process* of the net system. The net system is referred to as the *originative* system of the branching process. Note that the unfolding procedure of a net system can be stopped at different times yielding different branching processes.

The relation between a net system and its branching processes technically builds on a homomorphism between two nets that preserves the nature of nodes and the environment of transitions. Let $N_1 = (P_1, T_1, F_1)$ and $N_2 = (P_2, T_2, F_2)$ be two nets. A *homomorphism* from N_1 to N_2 is a mapping $h : P_1 \cup T_1 \to P_2 \cup T_2$, such that: $h(P_1) \subseteq P_2$ and $h(T_1) \subseteq T_2$ (the nature of nodes is preserved), and for all $t \in T_1$, the restriction of h to $\bullet t$ is a bijection between $\bullet t$ in N_1 and $\bullet h(t)$ in N_2; correspondingly for $t\bullet$ and $h(t)\bullet$ (the environment of transitions is preserved).

Definition 5.15 (Branching process).
A *branching process* of a net system $S = (N, M_0)$ is a pair $\beta = (N', \nu)$, where $N' = (B, E, G)$ is an occurrence net and ν is a homomorphism from N' to N, s.t.:

- the restriction of ν to $Min(N')$ is a bijection between $Min(N')$ and M_0, and
- for all $e_1, e_2 \in E$ holds, if $\bullet e_1 = \bullet e_2$ and $\nu(e_1) = \nu(e_2)$ then $e_1 = e_2$. ⌟

Two branching processes of the same net system are in a prefix relation. The prefix relation reflects that one branching process "unfolds" originative system less than another branching process. This is captured in the next definition.

Definition 5.16 (Prefix relation).
Let $\beta_1 = (N_1, \nu_1)$ and $\beta_2 = (N_2, \nu_2)$ be two branching processes of a net system $S = (N, M_0)$. β_1 *is a prefix of* β_2 if N_1 is a subnet of N_2, such that:

- $Min(N_2)$ belongs to N_1,
- if a condition belongs to N_1, then its input event in N_2 also belongs to N_1 (if such an event exists),
- if an event belongs to N_1, then its input and output conditions in N_2 also belong to N_1, and
- ν_1 is the restriction of ν_2 to nodes of N_1. ⌟

A branching process β' *is an extension of* a branching process β if β is a prefix of β'. When we talk about the reachable markings of a branching process, we refer to the markings reachable from the natural initial marking of the underlying occurrence net. In the context of occurrence nets, the natural initial marking is the marking that puts one token in each minimal condition (a condition with the empty preset) and no tokens elsewhere. Please note that every reachable marking of a branching process represents a reachable marking of its originative system.

Figure 5.8 shows four branching processes of the net system in Figure 2.6(a). Figure 5.8 shows the *initial branching process* – a branching process composed of conditions that correspond to the places in the initial marking of the originative system and no events. Indices of conditions and events in the figures reflect the mapping of the respective nodes to places and transitions of the originative system, e.g., c_1 and c_1' in Figure 5.8(d) are conditions that correspond to place p_1 in Figure 2.6(a). Observe that as the originative system allows infinite firing sequences, it has an infinite number of branching processes. Finally, note that

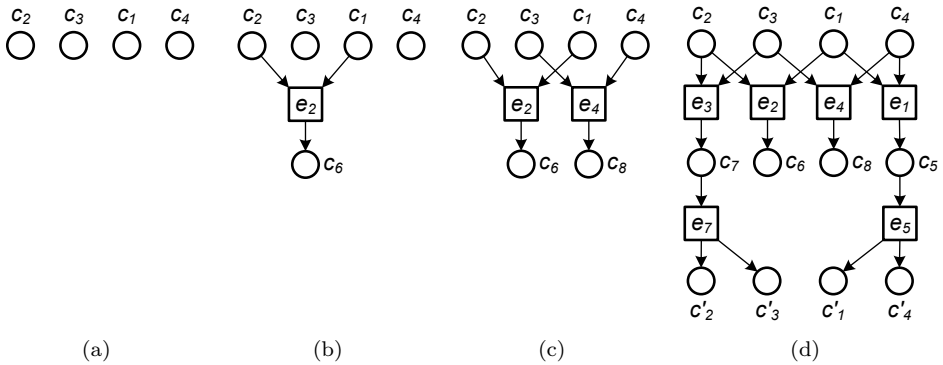


Figure 5.8. (a)–(d) Branching processes of the net system in Figure 2.6(a), where (a) is the initial branching process

branching processes in Figure 5.8 are in the prefix relation from left to right, i.e., for every pair of branching processes in the figure it holds that the branching process on the left is a prefix of the branching process on the right.

Next, we discuss notions which provide a convenient mechanism for explaining reachable markings of the originative system in terms of its branching processes.

Definition 5.17 (Configuration).
Let $\beta = (N, \nu)$, $N = (B, E, G)$, be a branching process of a net system.
A *configuration* C of β is a set of events, $C \subseteq E$, such that:

- C is causally closed, i.e., $e \in C$ implies that for all $e' \in E$, $e' \rightsquigarrow e$, holds $e' \in C$, and
- C is conflict-free, i.e., for all $e_1, e_2 \in C$ holds $\neg(e_1 \# e_2)$. ⌟

A configuration is a set of events in a branching process which represents a footprint of a firing sequence (or several firing sequences) of the originative system. In other words, for a given configuration there exists a firing sequence (or several firing sequences) of the originative system composed of the transitions that correspond to the events in the configuration. For instance, the set $\{e_1, e_3, e_5\}$ of events from Figure 5.8(d) is a configuration. This configuration represents several firing sequences of the originative system, e.g., t_1, t_3, t_5 (which is depicted in Figure 2.6) or t_3, t_1, t_5, etc. In contrast, $\{e_1, e_2\}$ and $\{e_5\}$ are *not* configurations of the branching process in Figure 5.8(d): $\{e_1, e_2\}$ is *not* conflict-free, $\{e_5\}$ is *not* causally closed.

Branching processes capture reachable markings of the originative system. A reachable marking M of a net system $S = (N, M_0)$, $N = (P, T, F)$, *is captured in* a branching process $\beta = (N', \nu)$ of S if β contains a cut such that for every place $p \in P$, the cut contains exactly $M(p)$ conditions that correspond to p. Recall from Section 5.2 that a *cut* is a maximal set of pairwise concurrent places of the net (or conditions in the context of occurrence nets). For instance, the marking $\{p_5, p_7\}$, see Figure 2.6(c), is captured in the branching process in Figure 5.8(d), as the branching process contains the cut $\{c_5, c_7\}$.

Every finite configuration of a branching process induces a cut.

Definition 5.18 (Cut induced by a finite configuration).
Let C be a finite configuration of a branching process $\beta = (N, \nu)$. Then, the co-set $(Min(N) \cup C\bullet) \setminus \bullet C$, denoted by $Cut(C)$, is a cut of N induced by C. ⌟

$Cut(C)$ is a set of conditions which are marked when all the events from configuration C fire (in any possible order starting from the natural initial marking of the underlying occurrence net). Given a finite configuration C of a branching process $\beta = (N, \nu)$ of an originative system S, the multi-set of places $\nu(Cut(C))$ is a reachable marking of S, which we denote by $Mark(C)$.

5.5.2. Unfoldings – Maximal Branching Processes

As already mentioned, the unfolding procedure can be stopped at different times yielding different branching processes. However, there is a unique (possibly infinite) branching process created by unfolding "as much as possible" [38]. The maximal branching process of a net system with respect to the prefix relation is called the *unfolding* of the net system. In [34], it is shown that every net system has a unique (up to isomorphism) unfolding. This section presents the unfolding algorithm from [38, Section 4.1]. Given a net system, the algorithm constructs its unfolding.

Javier Esparza, Stefan Römer, and Walter Vogler suggest implementing a branching process of a net system S as a set of nodes $\{n_1, \ldots, n_k\}$, each either a condition or an event. The authors propose encoding every condition as a record containing two fields: a place of S and a reference to the unique input event of the condition. If the condition has an empty preset, the reference is set to NULL (denoted by $\varnothing$). Thus, we denote a condition as a pair (p, e) or $(p, \varnothing)$, where p is a place of S and e is an event of unfolding. An event can also be captured as a record composed of two fields: a transition of S and a set of references to the input conditions of the event. In the following, an event is denoted as a pair (t, X), where t is a transition of S and X is the set of input conditions of the event. Please observe that the set of events and conditions (each captured as a record described above) also defines the flow relation of the unfolding and the mapping of its nodes to the nodes of the originative system.

The main idea of the unfolding algorithm is to start with the initial branching process, and then to iteratively add new nodes to the branching process. Observe that the initial marking is a multi-set and, thus, unfolding can contain several minimal conditions that correspond to the same place of the originative system. New events are added to the unfolding one at a time together with their output conditions. This iterative procedure requires a notion of a possible extension – an event that can be added to a branching process. Let t be a transition of the originative system with output places $\{p_1, \ldots, p_n\}$. An event $e = (t, X)$ is a possible extension of a branching process $\{n_1, \ldots, n_k\}$, if $\{n_1, \ldots, n_k, e, (p_1, e), \ldots, (p_n, e)\}$ is also a branching process. The set of all possible extensions of a branching process β is denoted by $PossibleExtensions(\beta)$.

In [38], the authors give the following characterization of a possible extension.

Proposition 5.1: *Let β be a branching process of a net system S and let ν be a mapping of nodes of β to nodes of S. The possible extensions of β are the pairs*

(t, X), *where* t *is a transition of* S *and* X *is a co-set of conditions of* β, *such that* $\nu(X) = {}^\bullet t$ *and* (t, X) *does not already belong to* β. ⋆

Considering all of the above, Algorithm 3 summarizes a technique for the construction of the unfolding of a given net system.

Algorithm 3: The unfolding algorithm [38]

Input: A net system $S = (N, M_0)$, where $M_0 = \{p_1, \ldots, p_n\}$
Output: The unfolding Unf of S

```
1 Unf = {(p_1, ∅), ..., (p_n, ∅)}
2 pe = PossibleExtensions(Unf)
3 while pe ≠ ∅ do
4 |   Add to Unf an event e = (t, X) ∈ pe and a condition (p, e) for every
  |   output place p of t
5 |   pe = PossibleExtensions(Unf)
6 return Unf
```

Observe that the algorithm terminates if and only if the originative net system S does not contain any infinite firing sequence. Note that unfoldings of the net systems in Figure 2.6 are infinite.

5.5.3. Finite Complete Prefix Unfoldings

Every marking captured in a branching process is a reachable marking of its originative system, and every reachable marking of a net system is captured in its unfolding [38]. A complete branching process of a net system is usually smaller than its unfolding (it is a finite prefix of the unfolding), but captures all the information about reachable markings of the net system.

Definition 5.19 (Complete branching process).
Let $\beta = (N', \nu)$, $N' = (B, E, G)$, be a branching process of a net system $S = (N, M_0)$, $N = (P, T, F)$. β is *complete* if for every reachable marking M of S there exists a configuration C of β, such that:

- M is captured in β, i.e., $Mark(C) = M$, and
- for every transition $t \in T$ enabled at M in N, there exists a configuration $C \cup \{e\}$, $e \in E$, of β such that $e \notin C$ and $\nu(e) = t$. ⌟

The algorithm from [38] for construction of a complete prefix unfolding (a complete branching process) of a net system is founded on the three notions of local configuration, adequate order, and cutoff event. Next, we discuss these notions.

Kenneth L. McMillan [84] proposed to associate every event of a branching process with a reachable marking of the originative system. The association is implemented with the help of the intermediate notion, viz. *local configuration*.

Definition 5.20 (Local configuration).
Let $\beta = (N, \nu)$, $N = (B, E, G)$, be a branching process of a net system.
The local configuration $\lceil e \rceil$ of an event $e \in E$ of β is the set of events $e' \in E$ such that $e' = e$ or $e' \rightsquigarrow e$, i.e., $\lceil e \rceil = \{e\} \cup \{e' \in E \mid e' \rightsquigarrow e\}$. ⌟

Algorithm 4: The complete prefix unfolding algorithm [38]

Input: A bounded net system $S = (N, M_0)$, where $M_0 = \{p_1, \ldots, p_n\}$
Output: A complete finite prefix Fin of S

$Fin = \{(p_1, \varnothing), \ldots, (p_n, \varnothing)\}$
$pe = PossibleExtensions(Fin)$
$cutoff = \varnothing$
while $pe \neq \varnothing$ **do**
  Choose an event $e = (t, X) \in pe$, s.t. $\lceil e \rceil$ is minimal with respect to $\lhd$
  if $\lceil e \rceil \cap cutoff = \varnothing$ **then**
    Add to Fin the event e and a condition (p, e) for every output place p of t
    $pe = PossibleExtensions(Fin)$
    if *e is a cutoff event of Fin* **then** $cutoff = cutoff \cup \{e\}$
  else $pe = pe \setminus \{e\}$
return Fin

Every event e of a branching process can be associated with the marking $Mark(\lceil e \rceil)$. Let us assume that during the construction of the unfolding an event e is added after event e', such that $Mark(\lceil e \rceil) = Mark(\lceil e' \rceil)$. If this is the case, it is reasonable to "cut-off" the construction of the unfolding either at event e or e' (recall that the unfolding of a net system is unique and, thus, branching processes induced by $Mark(\lceil e \rceil)$ and $Mark(\lceil e' \rceil)$ are isomorphic). The choice between e and e' is carried out based on a partial order, viz. *adequate order*.

Definition 5.21 (Adequate order).
A partial order $\lhd$ on the finite configurations of a branching process β is an *adequate order*, if (let C_1 and C_2 be two finite configurations of β):

- $\lhd$ is well-founded,
- $C_1 \subset C_2$ implies $C_1 \lhd C_2$, and
- if $C_1 \lhd C_2$ and $Mark(C_1) = Mark(C_2)$, then $\lhd$ is preserved for all finite extensions of C_1, cf., [38] for details. ⌟

The usage of an adequate order when deciding at which event to stop the construction of the unfolding guarantees the resulting branching process to be complete.

Definition 5.22 (Cutoff event).
Let $\beta = (N, \nu)$, $N = (B, E, G)$, be a branching process and let $\lhd$ be an adequate order on the finite configurations of β. An event $e \in E$ is a *cutoff* event of β, induced by $\lhd$, iff there exists a *corresponding* event, denoted by $corr(e) \in E$, such that $Mark(\lceil e \rceil) = Mark(\lceil corr(e) \rceil)$ and $\lceil corr(e) \rceil \lhd \lceil e \rceil$. ⌟

Finally, a *complete prefix unfolding* of a net system S is a special branching process of S which is obtained by truncating its unfolding at cutoff events.

Definition 5.23 (Complete prefix unfolding).
Let β be the unfolding of a net system S and let $\lhd$ be an adequate order on

the finite configurations of β. The *complete prefix unfolding* of S, induced by $\lhd$, is the maximal prefix of β with respect to the prefix relation that contains no event after a cutoff event of β induced by $\lhd$. ⌟

Algorithm 4 summarizes a technique for the construction of complete prefix unfoldings of net systems. The algorithm defines a family of algorithms – it can be instantiated using different adequate orders. Characteristics of the adequate order used in the algorithm determine the characteristics of the resulting complete prefix unfoldings. The adequate order is employed to select new events to be added to the complete prefix unfolding and to identify cutoff events. The algorithm terminates when there are no events to add. In the following, we discuss and compare two adequate orders: the adequate order for bounded systems proposed by Kenneth L. McMillan [84] and the adequate total order for safe systems proposed by Javier Esparza et al. [37, 38].

McMillan's Adequate Order

In his seminal work [84], Kenneth L. McMillan showed that a simple order on local configurations, when one local configuration is smaller than the other if and only if it contains less events, leads to a construction of a finite complete prefix unfolding for an arbitrary bounded net system. This is captured in the next definition.

Definition 5.24 (Adequate order for bounded systems).
Let $\beta = (N, \nu)$, $N = (B, E, G)$, be a branching process of a net system. Let $e_1, e_2 \in E$ be two events of β. Then, $\lceil e_1 \rceil \lhd_M \lceil e_2 \rceil$ holds, if $|\lceil e_1 \rceil| < |\lceil e_2 \rceil|$. ⌟

It is easy to see that order $\lhd_M$ is an adequate order.

Adequate Total Order for Safe Systems

It is highly preferable that an adequate order is also total. If one instantiates Algorithm 4 with an adequate total order $\lhd$, then whenever a new event e is generated after event e' such that $Mark(\lceil e \rceil) = Mark(\lceil e' \rceil)$, it holds that $\lceil e \rceil \lhd \lceil e' \rceil$ and, thus, e is identified as a cutoff event. Note that not every adequate order is a total order. For instance, McMillan's adequate order is not total. The existence of an adequate total order for arbitrary net systems is an open problem. However, it is known that there exists an adequate total order for the subclass of safe net systems (proposed by Javier Esparza et al. in [37, 38]).

Adequate total order for safe systems technically builds on the notions of quasi Parikh vectors over sets of events and Foata normal forms of configurations. These are discussed next.

Definition 5.25 (Quasi Parikh vector of events).
Let $\beta = (N', \nu)$, $N' = (B, E, G)$, be a branching process of a net system $S = (N, M_0)$, $N = (P, T, F)$. Let $\ll$ be an arbitrary total order on T. A *quasi Parikh vector* over a set of events $H \subseteq E$ with respect to $\ll$, denoted by $\chi^{\ll}(H)$, is a sequence of transitions which is ordered according to $\ll$, and contains transition $t \in T$ as many times as there are events in H that correspond to t, i.e., transition t appears in the sequence $|\{h \in H \mid \nu(h) = t\}|$ times. ⌟

Consider a set of events $H = \{e_1, e_2, e_2', e_3\}$ such that e_1 corresponds to transition t_1, e_2 and e_2' correspond to t_2, and e_3 corresponds to t_3. Let us fix an arbitrary total order $\ll$ on transitions, e.g., $t_3 \ll t_1 \ll t_2$. Then, the quasi Parikh vector over H is the sequence t_3, t_1, t_2, t_2.

Definition 5.26 (Order on quasi Parikh vectors).
Let $\beta = (N', \nu)$, $N' = (B, E, G)$, be a branching process of a net system $S = (N, M_0)$, $N = (P, T, F)$. Let $\ll$ be a total order on T and let $E_1 \subseteq E$ and $E_2 \subseteq E$ be two sets of events. Then, $\chi^{\ll}(E_1) \ll \chi^{\ll}(E_2)$ holds, if $\chi^{\ll}(E_1)$ is lexicographically smaller than $\chi^{\ll}(E_2)$ with respect to the order $\ll$. ⌟

Given a finite configuration of a branching process, Algorithm 5 constructs its Foata normal form, which is a sequence of sets of events of the configuration obtained by iteratively tearing off its minimal events.

Algorithm 5: Foata normal form of a configuration [38]

Input: A configuration C of a branching process
Output: The Foata normal form FC of C

```
1 FC = ∅
2 while C ≠ ∅ do
3 |   Append Min(C) to FC
4 |   C = C ∖ Min(C)
5 return FC
```

Please note that Algorithm 5 allows for the overload of notation and uses $Min(C)$ to denote the set of minimal events of configuration C with respect to the causal relation implied by the branching process.

Definition 5.27 (Order on Foata normal forms).
Let β be a branching process of a net system $S = (N, M_0)$, $N = (P, T, F)$. Let $\ll$ be a total order on T and let C_1 and C_2 be two configurations of β with Foata normal forms $FC_1 = C_{11}, \ldots, C_{1n_1}$ and $FC_2 = C_{21}, \ldots, C_{2n_2}$, respectively.
Then, $FC_1 \ll FC_2$ holds, if there exists $1 \leq i \leq n_1$ such that:

- $\chi^{\ll}(C_{1j}) = \chi^{\ll}(C_{2j})$, for every $1 \leq j < i$, and
- $\chi^{\ll}(C_{1i}) \ll \chi^{\ll}(C_{2i})$. ⌟

Now, we can define an adequate total order.

Definition 5.28 (Adequate total order for safe systems).
Let β be a branching process of a net system $S = (N, M_0)$, $N = (P, T, F)$. Let $\ll$ be a total order on T and let C_1 and C_2 be two configurations of β.
Then, $C_1 \lhd_E C_2$ holds, if:

- $|C_1| < |C_2|$, or
- $|C_1| = |C_2|$ and $\chi^{\ll}(C_1) \ll \chi^{\ll}(C_2)$, or
- $\chi^{\ll}(C_1) = \chi^{\ll}(C_2)$ and $FC_1 \ll FC_2$. ⌟

Please refer to [38] for the proof that $\lhd_E$ is indeed an adequate total order for the class of safe net systems.

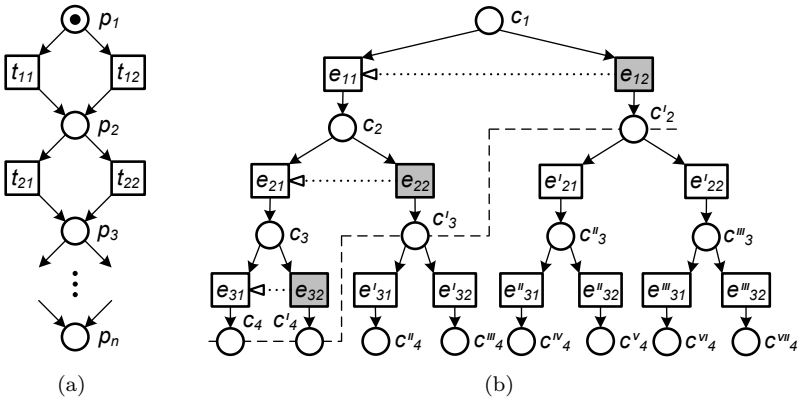


Figure 5.9. (a) A family of net systems, (b) two complete prefix unfoldings of the net system from the family (a) with respect to $\lhd_M$ and $\lhd_E$ adequate orders ($n = 4$)

Figure 5.9 exemplifies application of Algorithm 4. Figure 5.9(b) shows the complete prefix unfolding of the net system ($n = 4$) from the family of net systems in Figure 5.9(a) with respect to $\lhd_E$ adequate total order (see the subnet induced by nodes that are above or intersect the dashed line) superposed on the complete prefix unfolding of the same system with respect to $\lhd_M$ adequate order (see the whole net). Note that the complete prefix unfolding with respect to $\lhd_M$ is equal to the unfolding of the system. Events with the grey background are the cutoff events induced by $\lhd_E$. In the figure, dotted arrows show the relation between cutoff events and corresponding events, e.g., e_{11} is the corresponding event of cutoff event e_{12}, etc. Note that events e_{12}, e_{22}, and e_{32} cannot be identified as cutoff events with respect to $\lhd_M$. Consider the pair of events e_{11} and e_{12}. It holds, $Mark(\lceil e_{11} \rceil) = \{p_2\} = Mark(\lceil e_{12} \rceil)$. However, it also holds that $|\lceil e_{11} \rceil| = |\lceil e_{12} \rceil|$.

In [38], the authors also propose an adequate order for arbitrary net systems. Similar as $\lhd_E$, this adequate order tends to deliver more compact complete prefix unfoldings as compared to $\lhd_M$ (also in the case of unsafe net systems). In fact, this adequate order can deliver the same complete prefix unfolding (up to isomorphism) as the one provided by $\lhd_E$ in the example from Figure 5.9. However, this adequate order is not total – neither for arbitrary net systems, nor for the safe ones. Please refer to [38] for further details.

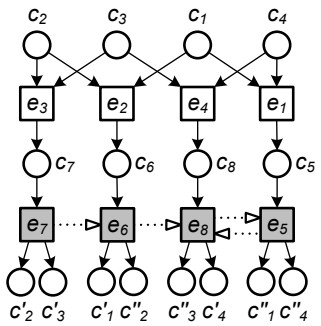
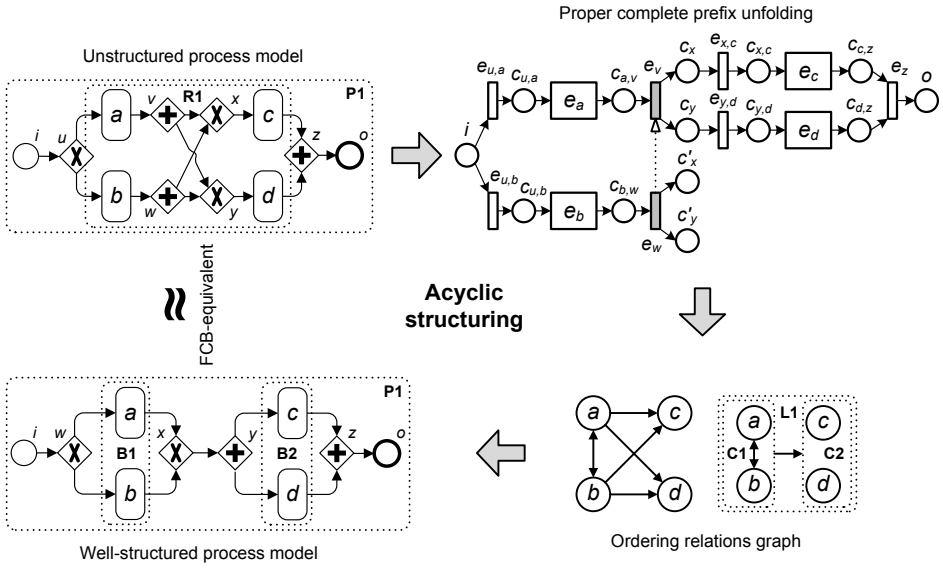


Figure 5.10. The complete prefix unfolding of the net system in Figure 2.6(a)

Figure 5.10 exemplifies the complete prefix unfolding of the net system in Figure 2.6(a). Please note that the complete prefix unfolding is obtained by using a slightly modified version of Algorithm 4 and the adequate total order for safe systems. In every iteration of the main loop of the algorithm

(lines 4–10), we performed additional checks on every newly added event e (line 7). When performing the checks, we waited for the first two events for which holds that $Mark([e])$ is equal to the initial marking of the originative system. Once identified, these first two events were immediately recorded as a cutoff event and its corresponding event. Otherwise, Algorithm 4 constructs a larger complete prefix unfolding than the one proposed in Figure 5.10. This larger complete prefix unfolding contains an event which is associated with the initial marking of the originative system and that is then used as a corresponding event of cutoff events that are also associated with the initial marking of the originative system (verify this by constructing the complete prefix unfolding of the net system in Figure 2.6(a) using Algorithm 4 and the adequate total order for safe systems). The above described instructions can be trivially injected into Algorithm 4 after line 9. Observe that the above suggested extension of the algorithm is of no use when unfolding WF-systems, as the initial marking of a WF-system is only reachable from its initial marking via the empty firing sequence.

6. Structuring Techniques



This chapter proposes several techniques for structuring process models: Section 6.1 presents a method for structuring acyclic models with single source and single sink nodes. Section 6.2 extends the technique from Section 6.1 to allow maximal structuring. The extension addresses the cases in which process models have no well-structured versions but can still be partially structured. Section 6.3 discusses particularities of structuring acyclic process models with multiple source and/or multiple sink nodes. Afterwards, Section 6.4 discusses ideas towards a technique for structuring cyclic process models. Finally, Section 6.5 draws conclusion and discusses future steps in the research on structuring process models.

The materials reported in this chapter are partially published in [106, 107, 108].

6.1. Acyclic Structuring

This section proposes an approach for structuring acyclic process models. Figure 6.1 summarizes the approach by showing the chain of phases that collectively compose the structuring technique. In the following, we assume that every process model has a single source node and a single sink node and that every gateway node can be classified as either a split or a join.

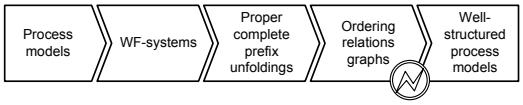


Figure 6.1. Acyclic structuring chain

First, a process model gets decomposed into the hierarchy of its process components. Each component is a process model by itself and either well-structured or unstructured. An unstructured process component can in some cases be transformed into a well-structured one. For this purpose, the component is translated into a WF-system for which the ordering relations of its tasks are derived from its proper complete prefix unfolding. If the ordering relations have certain properties, the unstructured component can be replaced by a hierarchy of smaller well-structured components that define the same ordering relations. Note that the equivalence of the resulting process model with the original unstructured model is guaranteed due to the results presented in [143].

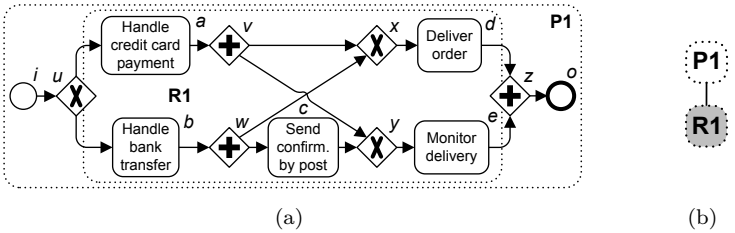


Figure 6.2. Unstructured acyclic process model

In the following, we discuss each phase of the acyclic structuring technique in detail. We employ the process model in Figure 6.2 to exemplify all structuring phases. Figure 6.2(a) shows the process model, while Figure 6.2(b) shows its RPST as a tree-like structure where the only rigid component $R1$ is highlighted with grey background. Later, in Section 6.2, we shall extend the technique to allow maximal structuring.

6.1.1. From Process Models to Unfoldings

A process model is well-structured if and only if its RPST contains no rigid component (Definition 5.3). Therefore, an unstructured process model can be structured by traversing its RPST bottom-up and replacing each rigid component

with its equivalent well-structured component. The difficult step is to find this equivalent well-structured component.

The key idea of structuring is to *refine* a rigid component R, i.e., a *node* of the RPST, with a subtree of well-structured RPST nodes which define the same behavioral (ordering) relations between the child RPST nodes of R. The first step for refining R is to compute the ordering relations of R's child nodes. We obtain these by constructing a complete prefix unfolding of R's corresponding WF-system. The complete prefix unfolding captures information about all reachable markings of the originative system, but has a simpler structure as it is an occurrence net (Definition 5.12). To capture all well-structuredness contained in R, the complete prefix unfolding must have a specific shape, called *proper*.

Definition 6.1 (Proper complete prefix unfolding).
Let $\beta = (N, \nu)$, $N = (B, E, G)$, be a branching process of a net system S and let $\lhd$ be an adequate order on the finite configurations of β.

- A cutoff event $e \in E$ of β induced by $\lhd$ is *healthy*, iff $Cut(\lceil e \rceil) \setminus e\bullet = Cut(\lceil corr(e) \rceil) \setminus corr(e)\bullet$.
- β is the *proper complete prefix unfolding*, or the *proper prefix*, of S induced by $\lhd$, iff β is the maximal prefix of the unfolding of S that contains no event after a healthy cutoff event induced by $\lhd$. ⌟

A proper prefix contains all information about well-structuredness, i.e., all paired gateways of splits and joins, in a rigid in the following way: A proper prefix β represents each *xor* split as a condition with multiple post-events; each *xor* join is identified by the post-conditions of a cutoff event e and its corresponding event. The notion of a cutoff event guarantees that β contains every *xor* split and join. An important observation here is that corresponding pairs of *xor* splits and joins are always contained in the same branch of β. An *and* split manifests as an event with multiple post-conditions, whereas an *and* join is an event with multiple pre-conditions. The healthiness requirement on cutoff events ensures that concurrency after an *and* split is kept encapsulated: when several concurrent branches are introduced in the unfolding they are not truncated until the point of their synchronization, i.e., the *and* join is reached. Such an intuition supports our goal of deriving a well-structured process model, because bonds of a well-structured process model that define concurrency must be synchronized in the same branch of the process model where they forked.

Figure 6.3 shows (a) a WF-net that corresponds to rigid component $R1$ in Figure 6.2, (b) a complete prefix unfolding, and (c) a proper prefix of the WF-net in (a). Both the complete prefix unfolding and the proper prefix are induced by the $\lhd_E$ adequate order, see Section 5.5.3. In the complete prefix unfolding, event e_c is a cutoff event, whereas event e_v is its corresponding event. The local configurations of events e_c and e_v induce the marking $\{p_x, p_y\}$ represented by the cuts $\{c_x, c_y\}$ and $\{c'_x, c'_y\}$, respectively. The complete prefix unfolding is *not* proper. Cutoff event e_c is not healthy, as $Cut(\lceil e_c \rceil) \setminus e_c\bullet = \{c'_x\}$, while $Cut(\lceil e_v \rceil) \setminus e_v\bullet = \varnothing$. Observe that the only cutoff event e'_z in Figure 6.3(c) is healthy, this results in the proper prefix where all the concurrency introduced by event e_w is synchronized in the same branch of the prefix by event e'_z. Note that in this example the

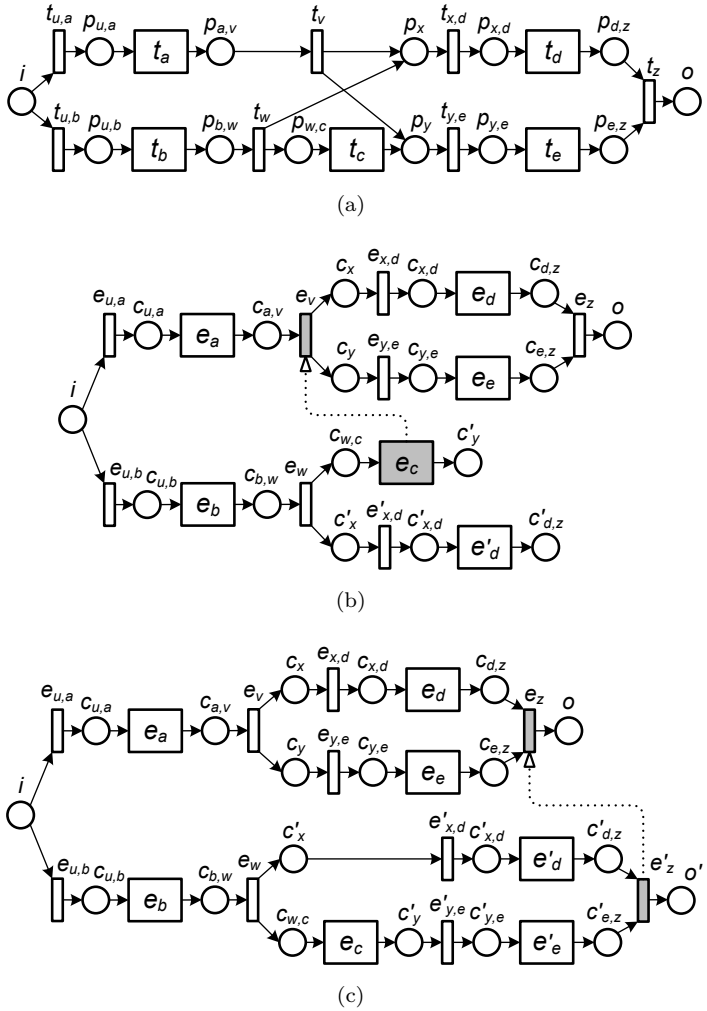


Figure 6.3. (a) A WF-net that corresponds to the rigid component $R1$ in Figure 6.2, (b) a complete prefix unfolding of (a), and (c) a proper prefix of (a)

proper prefix happens to be the unfolding of the system. This is not the general case. Figure 6.4 shows a proper prefix of the WF-net that corresponds to rigid component $R1$ in Figure 5.1(a), which is proper and smaller than the unfolding.

A proper complete prefix unfolding of an acyclic system is clearly finite. For structuring purposes, when computing proper prefixes we shall always use the $\lhd_E$ adequate order – the adequate total order for safe systems [37, 38]. For the class of safe systems, this adequate order results in minimal complete prefix unfoldings, if one only considers information about reachable markings induced by local configurations, which is exactly the case for healthy cutoff events. Recall that we only operate with sound and safe process models. Hence, when structuring acyclic process models, the $\lhd_E$ adequate order always yields minimal proper prefixes.

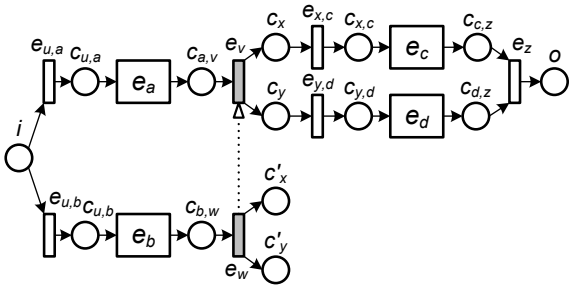


Figure 6.4. A proper prefix of the WF-net that corresponds to rigid component $R1$ in Figure 5.1(a) (subnet of the WF-net in Figure 5.4(a))

6.1.2. From Unfoldings to Graphs

Observable ordering relations of the proper prefix specify a unique behavioral footprint of its originative system. This section discusses an alternative representation of observable ordering relations. The relations get encoded in a directed graph – an *ordering relations graph*. In addition to being a convenient visualization mechanism, such a representation allows for the application of simpler algorithms for the analysis of ordering relations.

In order to overcome the effects of the proper prefix truncation at healthy cutoff events, we define the notion of proper ordering relations.

Definition 6.2 (Proper ordering relations).
Let $\beta = (N, \nu)$, $N = (B, E, G)$, be a proper prefix of an acyclic system and let $x, y \in B \cup E$ be its nodes. Let $\mathcal{R}_N = \{\rightsquigarrow_N, \nleftarrow_N, \#_N, \|_N\}$ be the ordering relations of N.

- x and y are in *proper causal* relation, written $x \twoheadrightarrow_N y$, iff $(x, y) \in G^+$ or there exists a sequence $(e_1, \ldots, e_n)$ of healthy cutoff events of β, $e_i \in E$, $1 \leq i \leq n$, $n \in \mathbb{N}$, such that $(x, e_1) \in G^*$, $(corr(e_i), e_{i+1}) \in G^*$ for $1 \leq i < n$, and $(corr(e_n), y) \in G^+$. y and x are in *inverse proper causal* relation, written $y \twoheadleftarrow_N x$, iff $x \twoheadrightarrow_N y$.
- The *proper conflict* relation of N is $\boxplus_N = \#_N \setminus (\twoheadrightarrow_N \cup \twoheadleftarrow_N)$.

The set $\mathcal{R}'_N = \{\twoheadrightarrow_N, \twoheadleftarrow_N, \boxplus_N, \|_N\}$ forms the *proper ordering relations* of N. ⌟

We omit subscripts of proper ordering relations where the context is clear. Similar to Definition 5.6, we refer to proper ordering relations $\mathcal{R}$ as *observable*, iff the relations in $\mathcal{R}$ only contain pairs of events that correspond to observable transitions. The (observable) (proper) ordering relations $\mathcal{R}$ define a two-structure $S = (N, R)$ (Section 2.2.1), where N is a set of events and R is an equivalence relation on $E_2(N)$, such that $e_1 \; R \; e_2$, $e_1, e_2 \in E_2(N)$, iff $e_1, e_2 \in \circledcirc$, $\circledcirc \in \mathcal{R}$. Figure 6.5(a) visualizes a two-structure defined by the observable proper ordering relations of the proper prefix shown in Figure 6.4. In the figure, we have arbitrarily chosen to encode proper conflict relation with solid edges, concurrent relation with dotted edges, proper causal relation with dashed edges, and inverse proper causal relation with dash-dotted edges.

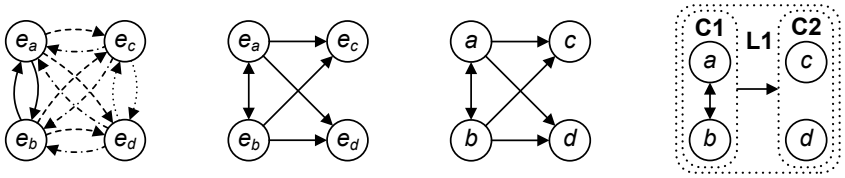


Figure 6.5. (a) Ordering relations two-structure, (b),(c) orgraphs, and (d) the MDT of (c)

The nature of ordering relations, i.e., conflict and concurrent relations are symmetric whereas the other two relations are the inverses of each other, results in the corresponding two-structure always being reversible (Section 2.2.2). Recall that the drawing of a reversible two-structure can be greatly simplified. Figure 6.5(b) shows one possible simplified drawing of the two-structure in Figure 6.5(a). To accomplish the drawing we have taken several design decisions: (i) Omit drawing inverse proper causal relation. (ii) Draw a solid two-sided arrow to encode proper conflict relation. (iii) Omit drawing concurrent relation. (iv) Draw a solid one-sided arrow to encode proper causal relation. The drawing in Figure 6.5(b) is a directed graph with vertices representing events of the proper prefix that correspond to observable transitions in the originative system. Figure 6.5(c) shows an alternative view to the drawing in which vertices represent transition labels.

We formally capture the design of the above described drawing approach in the notion of the *ordering relations graph*.

Definition 6.3 (Ordering relations graph).
Let $\beta = (N', \nu)$, $N' = (B, E, G)$, be a proper prefix of a labeled acyclic system $S = (N, M_0)$, $N = (P, T, F, \mathcal{T}, \lambda)$. Let $\{\rightarrowtail, \leftarrowtail, \boxplus, ||\}$ be the observable proper ordering relations of N'. An *ordering relations graph*, or *orgraph*, $\mathcal{G} = (V, A, \mathcal{B}, \sigma)$ of β has vertices $V \subseteq E$ defined by events of β that correspond to observable transitions of N, i.e., $V = \{e \in E \mid \lambda(\nu(e)) \neq \tau\}$, arcs $A = \rightarrowtail \cup \boxplus$, and a labeling function $\sigma : V \to \mathcal{B}$, $\mathcal{B} = \mathcal{T} \setminus \{\tau\}$ with $\sigma(v) = \lambda(\nu(v))$, $v \in V$. ⌟

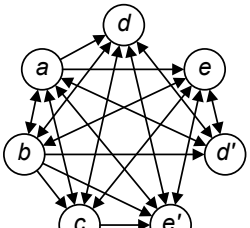


Figure 6.6. Ordering relations graph of the proper prefix in Figure 6.3(c)

The ordering relations graph of a proper prefix gives a structural characterization to the behavioral footprint of its originative system. Vertices of an orgraph represent events of occurrences of transitions from the originative system. The ordering relation between every pair of events can be uniquely determined based on edges that connect the events. Let $x, y \in V$ be two events of the orgraph $\mathcal{G} = (V, A, \mathcal{B}, \sigma)$. It then holds that: x and y are in proper causal relation if $(x, y) \in A \wedge (y, x) \notin A$. x and y are in inverse proper causal relation if $(x, y) \notin A \wedge (y, x) \in A$. x and y are in proper conflict relation if $(x, y) \in A \wedge (y, x) \in A$. Finally, x and y are in concurrent relation if $(x, y) \notin A \wedge (y, x) \notin A$.

Figure 6.6 visualizes the ordering relations graph of the proper complete prefix unfolding from Figure 6.3(c). Note that several events of a proper prefix can map to the same transition of its originative system, e.g., events e_d and e'_d in Figure 6.3(c) both map to transition t_d in the originative system. The same applies to orgraphs. The orgraph in Figure 6.6 contains two such pairs of events.

6.1.3. Parsing Two-Structures

In this section, we take a step back from the discussion of the technique for structuring acyclic process models and give the background on the technique for parsing two-structures [29, 31]. After the initial introduction to the theory of two-structures (see Section 2.2), this section shows how every two-structure can be constructed from (decomposed into) a hierarchy of two-structures of four basic classes. In the next section, we shall parse two-structures that encode ordering relations in order to finalize our structuring technique.

One of the central notions of the theory of two-structures is the notion of a *clan*.

Definition 6.4 (Clan).
A *clan* of a two-structure $S = (N, R)$ is a set $X \subseteq N$, such that for all $x, y \in X$ and for all $z \in N \setminus X$ holds $(z, x)\ R\ (z, y)$ and $(x, z)\ R\ (y, z)$. ⌟

Observe that for a reversible two-structure, it is sufficient that a weaker condition is satisfied in order to classify a subset of nodes of a two-structure as a clan [31].

Lemma 6.5: *Let $S = (N, R)$ be a reversible two-structure and let $X \subseteq N$. The following statements are equivalent:*

- *X is a clan of S,*
- *for all $x, y \in X$ and for all $z \in N \setminus X$ holds $(z, x)\ R\ (z, y)$, and*
- *for all $x, y \in X$ and for all $z \in N \setminus X$ holds $(x, z)\ R\ (y, z)$.* ⋆

Let $S = (N, R)$ be a two-structure. It immediately follows that $\varnothing$, $dom(S)$, and the singletons $\{n\}$, $n \in N$, are clans of S. These clans are the *trivial* clans of S. In the subsequent sections, we shall make extensive use of the important structural classes of two-structures, which are defined next.

Definition 6.6 (Linear, Complete, Primitive).
Let $S = (N, R)$ be a two-structure.

- S is *linear*, iff (i) S is asymmetric, (ii) for all distinct $x, y, z \in N$ holds either $(x, y)\ R^\star\ (x, z)$ or $(y, x)\ R^\star\ (y, z)$ or $(z, x)\ R^\star\ (z, y)$, (iii) for all $e_1, e_2 \in E_2(N)$ holds either $e_1\ R\ e_2$ or $e_1\ R\ e_2^{-1}$.
- S is *complete*, iff all its edges are of the same color.
- S is *primitive*, iff it contains at least three nodes and all clans in S are trivial. ⌟

Let $S = (N, R)$ with $|N| > 1$ and P be a partition of $E_2(N)$ induced by R. One can then alternatively define a complete and linear two-structure as follows: S is complete, iff $|P| = 1$. S is linear, iff $|P| = 2$ and there exists a linear order $(n_1, \ldots, n_{|N|})$ of elements of N such that the edges $\{(n_i, n_j) \mid i < j\}$ form an

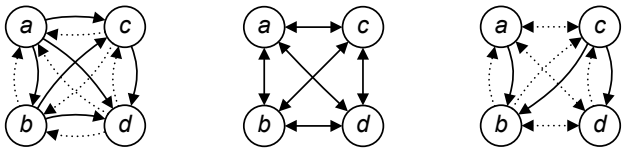


Figure 6.7. (a) A linear, (b) a complete, and (c) a primitive two-structure

equivalence class of R and the edges $\{(n_j, n_i) \mid i < j\}$ form an equivalence class of R, $1 \leq i, j \leq |N|$.

Figure 6.7 exemplifies different classes of two-structures: Figure 6.7(a) presents a linear two-structure that is defined by a linear order (a, b, c, d). Figure 6.7(b) demonstrates a complete two-structure where all edges of the two-structure form one equivalence class. Figure 6.7(c) shows a primitive two-structure where each proper subset of nodes consisting of more than one element is not a clan.

We denote by $\mathcal{C}(S)$ the set of all clans of S. The following result from [31] shows that restricting any investigations to reversible two-structures does not imply a loss of generality as far as the clans are concerned. We denote by S_X a substructure (or a *factor*) of a two-structure $S = (N, R)$ induced by a nonempty set $X \subseteq N$.

Theorem 6.7. *For each $S = (N, R)$ there exists a reversible two-structure $S' = (N, R')$, such that for all $X \subseteq N$ holds $\mathcal{C}(S_X) = \mathcal{C}(S'_X)$, i.e., $\mathcal{C}(S) = \mathcal{C}(S')$.* ⋆

Therefore, the sets of clans of S and $S^\star$ are equal, and as $S^\star$ is more restrictive, see Definition 2.5, and the conditions to be satisfied by a subset of nodes of $S^\star$ to be a clan are weaker, see Lemma 6.5, one can treat $S^\star$ as a normal form of S.

Every two-structure can be constructed from (decomposed into) a hierarchy of linear, complete, and primitive two-structures. Let $S = (N, R)$ be a (reversible) two-structure. Two subsets X and Y of the $dom(S)$ *overlap*, if $X \cap Y \neq \emptyset$, $X \setminus Y \neq \emptyset$, and $Y \setminus X \neq \emptyset$. The next results from [29] are essential for understanding the properties of such a hierarchy.

Lemma 6.8: *Let $S = (N, R)$ be a two-structure and let $X, Y \in \mathcal{C}(S)$. We have:*

- *$X \cap Y \in \mathcal{C}$.*
- *if X and Y overlap, then $X \cap Y$, $X \cup Y$, $X \setminus Y \in \mathcal{C}$.*
- *if $X \cap Y = \emptyset$, then $(x_1, y_1)R = (x_2, y_2)R$ for all $x_1, x_2 \in X$ and $y_1, y_2 \in Y$.* ⋆

Construction principles of a two-structure are defined by its decomposition into factors and a quotient that gives the relations between the factors.

Definition 6.9 (Factorization, Quotient, Decomposition).
Let $S = (N, R)$ be a two-structure.

- A partition $\chi = \{X_1, \ldots, X_k\}$, $k \in \mathbb{N}$, of N into nonempty clans is a *factorization* of S.
- The *quotient* of S by a factorization χ is a two-structure $S/\chi = (\chi, R_\chi)$, where $(X_1, Y_1)\ R_\chi\ (X_2, Y_2)$ iff $(x_1, y_1)\ R\ (x_2, y_2)$ for some $x_i \in X_i$, $y_i \in Y_i$, $X_i, Y_i \in \chi$.

- A *decomposition* $(S_{X_1}, \ldots, S_{X_k}; S/\chi)$ of S consists of the factors S_{X_1} with respect to a factorization $\chi = \{X_1, \ldots, X_k\}$ and the quotient S/χ. ⌟

Because of Lemma 6.8, the quotient S/χ is *well-defined*, i.e., it is independent of the choice of a representative of each clan from a partition. A nonempty clan X of S is *prime*, iff for every clan Y of S holds that X and Y do not overlap. We denote by $\mathcal{P}(S)$ the set of all prime clans of S. A prime clan is *maximal*, if it is maximal with respect to inclusion among *proper* prime clans of S, where a clan is proper if it is a proper subset of $dom(S)$. We denote by $\mathcal{P}_{max}(S)$ the set of all maximal prime clans of S; if $|N| = 1$, then $\mathcal{P}_{max}(S) = \{N\}$.

Theorem 6.10. *The maximal prime clans $\mathcal{P}_{max}(S)$ of a two-structure S form a partition of N and, therefore, the quotient $S/\mathcal{P}_{max}(S)$ is well-defined.* ⋆

Theorem 6.10 from [29] states that the domain of each two-structure can be partitioned by the domains of its maximal prime clans, whereas the following result states that the quotient of such a partition is a two-structure of one of three classes.

Theorem 6.11 (Clan Decomposition Theorem). *For each two-structure S, the quotient $S/\mathcal{P}_{max}(S)$ is either linear, or complete, or primitive.* ⋆

By iteratively discovering maximal prime clans and deriving the quotient for each factor that corresponds to an element of the decomposition, one builds a hierarchy of quotients. Such a hierarchy is unique for a given two-structure and can be seen as its structural characterization.

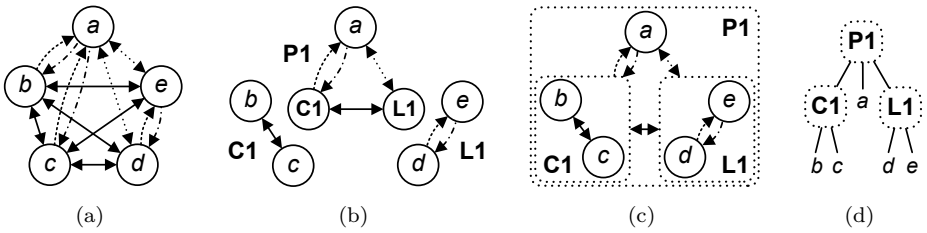


Figure 6.8. (a) A two-structure, and (b)–(d) decomposition of (a)

Figure 6.8 exemplifies the decomposition of a two-structure. Figure 6.8(a) shows a two-structure which is composed of five nodes and has four equivalence classes on its edges. Partition $\chi = \{\{a\}, \{b, c\}, \{d, e\}\}$ is the factorization of this two-structure. Two-structures induced by subsets of nodes $\{a\}$, $\{b, c\}$, and $\{d, e\}$ are, respectively, a trivial, complete, and linear clan of the original two-structure. Clan $P1$, see Figure 6.8(b), is the quotient of the two-structure by factorization χ; recall that clan names hint at their class, i.e., $P1$ is primitive, $C1$ is complete, and $L1$ is linear. Finally, Figure 6.8(c) organizes clans in a hierarchy in which each clan is enclosed in a dotted box with rounded corners, whereas containment of boxes represents the parent-child relation; Figure 6.8(d) gives an alternative tree-like representation of the hierarchy.

6.1.4. From Graphs to Process Models

The RPST of a well-structured process model is composed of trivial, polygon, and bond (either *and* or *xor*) components. In contrast to a rigid component, each component in a well-structured model has a well-defined and regular structure within the corresponding orgraph, which allows for a precise characterization. The ordering relations graph of a *xor* bond is a complete graph, or a clique, whereas the orgraph of an *and* bond is an edgeless graph. These topologies are consistent with the intuition behind them, i.e., all tasks in a *xor* bond are in conflict, i.e., only one is executed, and all tasks in an *and* bond are concurrently executed. Figure 6.9(a) shows a *xor* bond with three branches, whereas Figure 6.9(b) shows the corresponding complete orgraph. Similarly, Figure 6.9(c) and Figure 6.9(d) show an *and* bond and the corresponding orgraph, respectively. For trivial and polygon components, the orgraph is a direct acyclic graph representing the transitive closure, or the total order, of the proper causal relation. Figure 6.9(e) shows a polygon composed of three tasks, whereas Figure 6.9(f) presents the corresponding transitive closure over the proper causal relation.

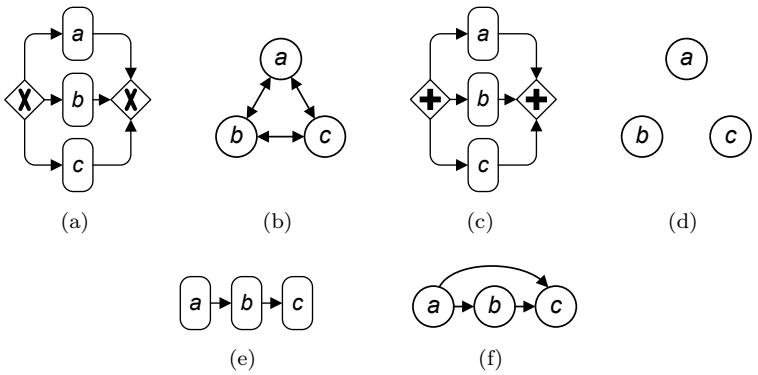


Figure 6.9. (a) A *xor* bond component, (b) a complete orgraph, (c) an *and* bond component, (d) an edgeless orgraph, (e) a polygon component, and (f) a total order orgraph

The main idea of our structuring technique is to parse a given orgraph into subgraphs. If one can decompose an orgraph into subgraphs such that every subgraph is either complete, edgeless, or total order, then one can construct a corresponding well-structured process component for each of the discovered subgraphs and, in this way, synthesize a well-structured process model.

To perform the decomposition, we rely on the parsing technique for two-structures which was discussed in Section 6.1.3. Orgraphs are special representations of two-structures which encode ordering relations, refer to Section 6.1.2. Due to the design of orgraphs, the parsing principles can be simplified. The clan decomposition of two-structures is closely related to the *modular decomposition* of (directed) graphs. Effectively, the parsing of an orgraph can be implemented by performing its *modular decomposition*. The technique of modular decomposition has been discovered independently by researchers in graph theory, network theory, and game theory. In our setting, modular decomposition is the clan decomposition

technique applicable to the restricted class of two-structures – orgraphs. A number of algorithms for performing modular decomposition exist; [20] gives a summary on the history of algorithms of the modular decomposition, whereas [90, 91] summarize works from different fields that employ modular decomposition.

Modular decomposition is a technique for parsing directed graphs into modules. Let $\mathcal{G} = (V, A, \mathcal{B}, \sigma)$ be an ordering relations graph. A *module* $M \subseteq V$ of $\mathcal{G}$ is a non-empty subset of vertices of $\mathcal{G}$ that are in uniform relation with vertices $V \setminus M$, i.e., if $v \in V \setminus M$, then v has directed arcs to all members of M or to none of them, and all members of M have directed arcs to v or none of them do. However, $v_1, v_2 \in V \setminus M$, $v_1 \neq v_2$, can have different relations to members of M. Moreover, the members of M and $V \setminus M$ can have arbitrary relations to each other [82]. For example, $\{a, b\}$ is a module in Figure 6.5(c), as both a and b are in proper causal relation with c and d. The above definition of a module supports our intent of synthesizing a process component from the ordering relations captured in a module; all tasks inside a SESE process component are in the same ordering relation with a given task outside the component.

Two modules M_1 and M_2 of $\mathcal{G}$ *overlap*, iff they intersect and neither is a subset of the other, i.e., $M_1 \setminus M_2$, $M_1 \cap M_2$, and $M_2 \setminus M_1$ are all non-empty. M_1 is *strong*, iff there exists no module M_2 of $\mathcal{G}$, such that M_1 and M_2 overlap. The *modular decomposition* substitutes each strong module of a graph with a new vertex and proceeds recursively (this recursive principle is discussed in detail for the case of two-structures in Section 6.1.3). The result is a unique rooted tree, called the *modular decomposition tree*, which can be computed in linear time [82].

Definition 6.12 (Modular Decomposition Tree).
Let $\mathcal{G}$ be an ordering relations graph. The *Modular Decomposition Tree* (MDT) of $\mathcal{G}$ is the set of all strong modules of $\mathcal{G}$. ⌟

According to [82], each module in the MDT belongs to one out of four classes.

Definition 6.13 (Trivial, Linear, Complete, Primitive).
Let M be a module of an ordering relations graph $\mathcal{G}$.

- M is a *trivial* module, iff M is singleton, i.e., M contains a single vertex.
- M is a *linear* module, iff there exists a linear order $(x_1, \ldots, x_{|M|})$ of elements of M, such that there is a directed arc from x_i to x_j in $\mathcal{G}$, iff $i < j$.
- M is a *complete* module, iff the subgraph of $\mathcal{G}$ induced by vertices in M is either complete or edgeless.
- M is a *primitive* module, iff M is neither a trivial, nor a linear, nor a complete module. ⌟

Please observe that module classes are just the "projections" of two-structure classes (see Section 6.1.3) on directed graphs. For our purposes, we classify modules further: If a complete module induces a complete subgraph, the module is referred to as a *xor* complete. If a complete module induces an edgeless subgraph, the module is referred to as an *and* complete. Finally, if a module induces a subgraph with a pair of distinct vertices which are not connected by an arc, the module is said to be *concurrent*.

Figure 6.10(a) shows the MDT of the ordering relations graph in Figure 6.6. Besides the trivial modules, the MDT contains linear modules $L1$, $L2$, and $L3$,

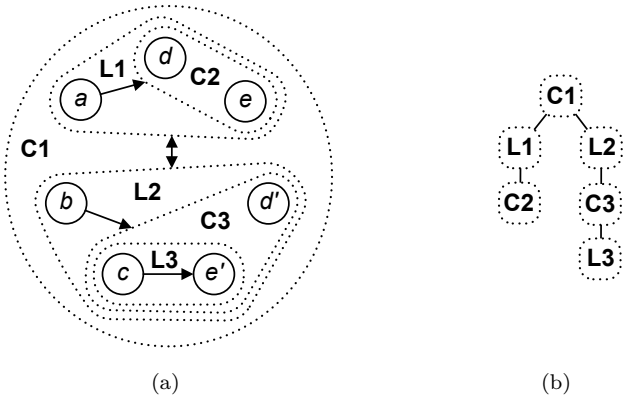


Figure 6.10. The MDT of the ordering relations graph in Figure 6.6

xor complete module $C1$, and *and* complete modules $C2$ and $C3$. Module $C1$ is the root module, whereas trivial modules are leafs of the MDT. In the figure, each area enclosed by a dotted line defines a module composed of the modules inside the area. Edges between modules represent relations between every pair of vertices composed of one vertex from each of the adjacent modules. Module names hint at their class, e.g., $C1$ is complete and $L1$ is linear. Figure 6.10(b) shows an alternative visualization of the MDT given as a tree-like structure, where nodes represent non-trivial modules and edges encode containment relation. Observe that we simplified the drawing by omitting trivial modules.

A rigid process component R of an RPST can be structured by refining R in the RPST to a subtree T_R. The root of T_R is a child of R's parent, each child of R is attached to a leaf of T_R, the nodes of T_R are defined by the modules of the MDT of R's ordering relations graph. The class of a node of T_R is determined by the characteristics of its defining MDT module.

We are now ready to present the main result of this section.

Theorem 6.14. *Let $\mathcal{G}$ be an ordering relations graph. The MDT of $\mathcal{G}$ contains no primitive module, iff there exists a well-structured process model PM such that $\mathcal{G}$ is the ordering relations graph of PM.* ⋆

The proof of Theorem 6.14 can be found in Appendix A.3. The proof of Theorem 6.14 implicitly specifies a procedure which, given a process model, synthesizes an FCB-equivalent well-structured model. This procedure is made explicit in Algorithm 6. Note that the algorithm expects a process model (process component) as input in which no pair of distinct tasks have the same label.

Without loss of generality, Algorithm 6 assumes that the RPST of the process model (or process component), taken as input, consists of a single rigid process component. The algorithm can be trivially extended to the case where the RPST of the process model given as input consists of a rigid component with polygons, bonds, or rigids as descendants. In this latter case, child components of the rigid component need to be abstracted into atomic task nodes by computing the extract of the rigid component, see Section 4.4. Once the extract is structured, its tasks

Algorithm 6: Structuring acyclic process model

Input: A sound acyclic process model PM
Output: An equivalent well-structured process model

```
1  Construct WF-net N that corresponds to PM
2  Construct proper complete prefix unfolding β of N
3  Construct ordering relations graph G of β
4  Compute M – the MDT of G
   // Construct process model PM' by traversing M in postorder
5  foreach module m of M do
6  |  switch class of m do
7  |  |  case m is trivial
8  |  |  |  Construct a task
9  |  |  case m is and complete
10 |  |  |  Construct an and bond
11 |  |  case m is xor complete
12 |  |  |  Construct a xor bond
13 |  |  case m is linear
14 |  |  |  Construct a trivial or polygon
15 |  |  case m is non-concurrent primitive
16 |  |  |  Construct a well-structured process component using compiler
   |  |  |  techniques, e.g., [97]
17 |  |  otherwise
18 |  |  |  FAIL
19 return PM'
```

must be refined back with the prior abstracted process components. Also, if we were given a process model whose RPST contains several rigid components, we would start by structuring the rigid components at lower levels of the RPST and collapsing them into atomic task nodes before attempting to structure rigids at upper levels. The complexity of the algorithm is determined by the complexity of the unfolding step. The computation of an orgraph has polynomial complexity. All other steps can be accomplished in linear time. As explained before, in the case of an *and* rigid, the unfolding step is not required, as nets which correspond to *and* rigids and their proper prefixes coincide. Note that the behavior captured in non-concurrent primitives can be structured by employing compiler techniques for Go To program transformations. To accomplish structuring, one first needs to synthesize a program (process component) from a non-concurrent primitive module. The synthesis can be trivially accomplished by adopting the technique in [33]. An acyclic process model has an equivalent well-structured model if its ordering relations graph contains no concurrent primitive module. The behavior captured by other module classes can be expressed by well-structured process components. Algorithm 6 traverses the MDT and constructs a process component for each encountered module from components that correspond to its child modules. A trivial module corresponds to a task. A linear module corresponds either to a

trivial component (if the module is the order on two elements) or to a polygon component (if the module is the order on more than two elements). An *and* (*xor*) complete module corresponds to a bond with *and* (*xor*) gateways as entry and exit nodes. Finally, a primitive module without concurrency can be structured using standard compiler techniques [97]. A well-structured process model constructed in such a way is fully concurrent bisimilar with the original unstructured model.

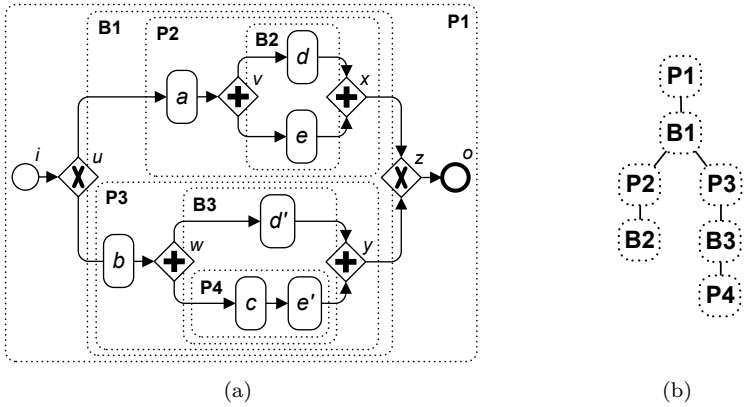


Figure 6.11. Well-structured acyclic process model

In our example, the RPST of the process model in Figure 6.2(a) contains one rigid $R1$. $R1$ in Figure 6.2(b) can be refined with a subtree of RPST nodes that correspond to modules of the MDT in Figure 6.10(b), resulting in the RPST shown in Figure 6.11(b). First, modules $C2$ and $L3$ should be used to construct components $B2$ and $P4$, respectively. Next, modules $L1$ and $C3$ must be employed to construct components $P2$ and $B3$. Afterwards, module $L2$ should result in component $P3$. Finally, module $C1$ results in component $B1$. The process model in Figure 6.11(a) is well-structured and FCB-equivalent to the process model in Figure 6.2(a). As another example, Figure 6.5(d) shows the MDT of the orgraph in Figure 6.5(c). Therefore, the well-structured version of the unstructured process model in Figure 5.1(a) shown in Figure 5.1(b) can be synthesized from the MDT in Figure 6.5(d). Modules $C1$, $C2$, and $L1$ in the MDT correspond to components $B1$, $B2$, and $P1$, respectively, in Figure 5.2(b).

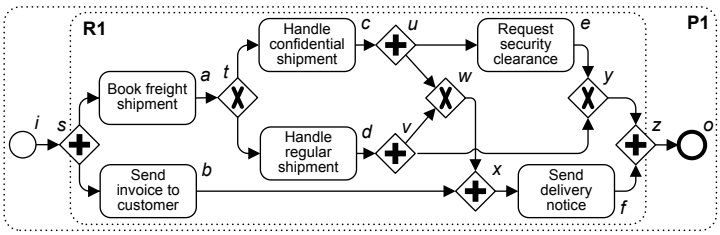
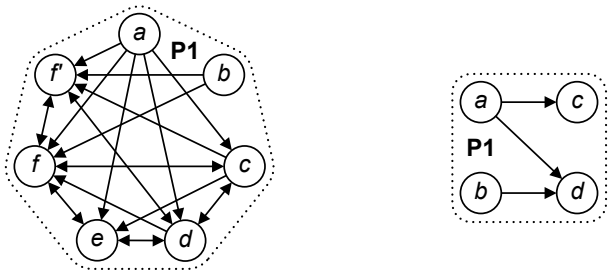


Figure 6.12. Inherently unstructured acyclic process model

Algorithm 6 fails if the input process model (process component) is inherently unstructured, such as component $R1$ in Figure 6.12. In this particular case, the ordering relations graph of $R1$ forms a single concurrent primitive module, refer to Figure 6.13(a). Please note that the unfolding step duplicates the transition which corresponds to task f. The duplication of tasks also takes place in the well-structured process model in Figure 6.11(a). As structuring relies on minimal proper prefixes (see the discussion in Section 6.1.1), the proposed structuring technique always operates with the minimum duplication of tasks which is required to allow structuring. For instance, the structuring of the process model in Figure 5.1(a) does not introduce any duplication of tasks, see Figure 5.1(b). Figure 6.13(b) shows the MDT of the orgraph of process component $R1$ in Figure 5.6. The MDT contains primitive module $P1$ which supports the idea that the process model is inherently unstructured.



(a) (b)

Figure 6.13. The MDTs of the orgraphs of process components $R1$ from the process models: (a) in Figure 6.12, and (b) in Figure 5.6

In light of the above, we conclude that given an acyclic process model with an arbitrary topology, one can construct an FCB-equivalent well-structured process model, if and only if the proper complete prefix unfolding of the system that corresponds to the given process model is such that the modular decomposition of its orgraph contains no concurrent primitive module.

6.2. Maximal Acyclic Structuring

According to Section 6.1, the process model in Figure 6.14 has no equivalent well-structured version – the evidence for this claim will be provided in Section 6.2.1. However, this does not answer the question of whether there exists an equivalent process model that is "better" structured than the model in Figure 6.14. In this section, we formalize the notion of a "better" structured version of a process model and propose a technique which given a process model constructs its maximally-structured version – an equivalent process model that has no "better" structured versions. We shall employ the model in Figure 6.14 as our running example.

We postpone a detailed introduction of the maximal acyclic structuring technique until Section 6.2.2. Meanwhile, we proceed with a formal definition of the maximally-structured property of process models.

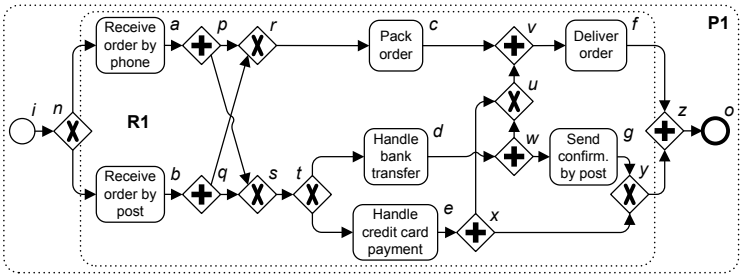


Figure 6.14. Unstructured acyclic process model

6.2.1. Maximally-structured Process Models

In this section, we give a formal definition of the maximally-structured property of process models. Unlike well-structuredness, maximal structuredness is founded on a relation *between the RPST and the MDT of the process model*. We gain insights into this relation by applying the structuring technique from Section 6.1 on the process model in Figure 6.14. To simplify language, in the following, we often skip several phases of the structuring technique when referring to the concepts of interest. For example, the "MDT of a process model" is the MDT obtained by mapping the model to a WF-net, constructing the proper prefix, then its ordering relations graph and, finally, computing the MDT of the graph.

Figure 6.15 visualizes transformation steps 3–18 of Algorithm 6 applied to the process model in Figure 6.14. The proper prefix of the model (shown at the top of the figure) is the result of line 2 of the algorithm. Next (follow the arrow directions), the algorithm constructs the ordering relations graph and its MDT (lines 3–4). Finally, the algorithm attempts to construct a process model from the MDT (lines 5–18). As the MDT contains primitive module $P2$, the algorithm fails at line 18. However, the MDT exhibits some information on structuredness of the ordering relations: it contains linear module $L1$ and *xor* complete module $C1$. These modules can be used to construct structured components $P4$ and $B1$, shown at the bottom of Figure 6.15. If one is able to synthesize a process component which demonstrates the ordering relations captured in primitive module $P2$, then one can construct a model which exhibits more structuredness than the original one. Note that in general a primitive module defines a subgraph of the ordering relations graph and, hence, the topology of the corresponding process component in the RPST cannot be deduced from the original model.

In a maximally-structured process model every module of its MDT must have a corresponding process component in its RPST.

Definition 6.15 (Maximally-structured process model).
A process model is *maximally-structured*, iff there exists isomorphism between the RPST and the MDT of the process model, where simple components of the RPST and trivial modules of the MDT are ignored, which assigns a linear module to every polygon component, a complete to every bond, a primitive to every rigid, and all primitives in the MDT are concurrent. ⌟

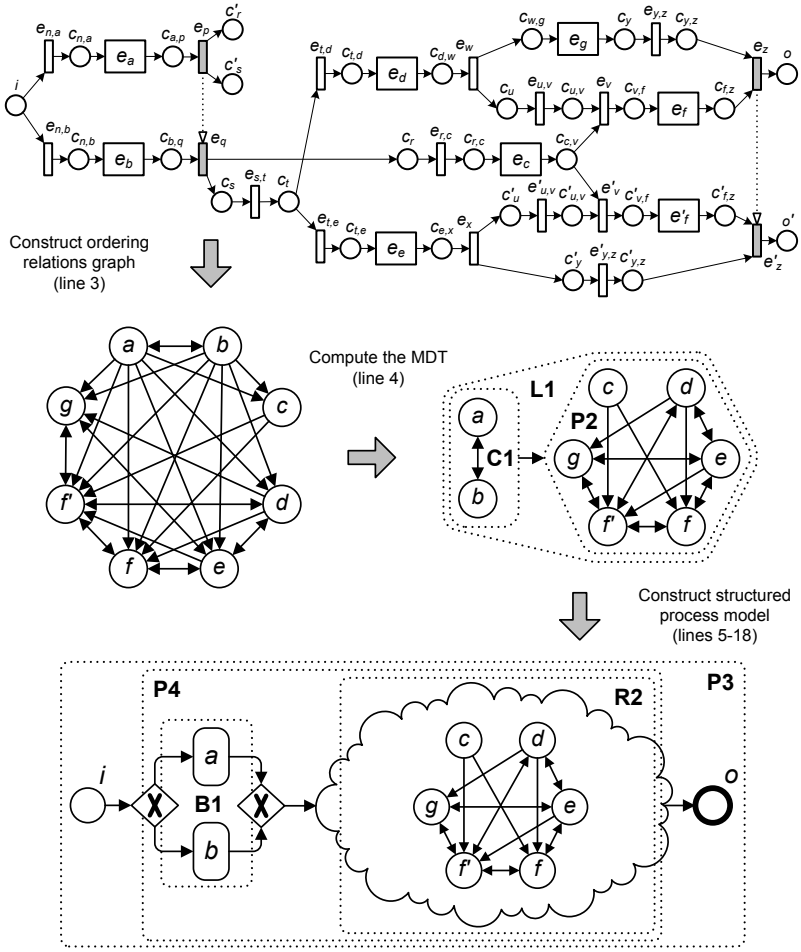


Figure 6.15. Structuring the process model in Figure 6.14

Recall that a simple component is either a trivial, or a polygon composed of two trivials (Section 5.1). Due to the properness of the complete prefix unfolding, the orgraph of a given process model and hence its MDT are unique. Consequently, whether a process model is maximally-structured is uniquely determined.

Figures 6.16(a) and 6.16(c) show the RPST of the model in Figure 6.14 and the subtree of the RPST of the model at the bottom of Figure 6.15, respectively. In Figure 6.16(b), one can see the MDT from the structuring scenario in Figure 6.15. This MDT is isomorphic to the RPST in Figure 6.16(c): Linear $L1$ can be mapped to polygon $P4$. Complete $C1$ can be mapped to bond $B1$. Primitive $P2$ can be mapped to rigid $R2$. In the final RPST of the maximally-structured version of Figure 6.14, polygon $P4$ must be merged with polygon $P3$ (as $P4$ is a subsequence of $P3$ and, thus, not canonical), see [113] for details. Note that Definition 6.15 is based on the implicit assumption that the isomorphism between both trees

canonically extends to an isomorphism that maps process components with tasks onto modules of events with the same multi-sets of names/labels.

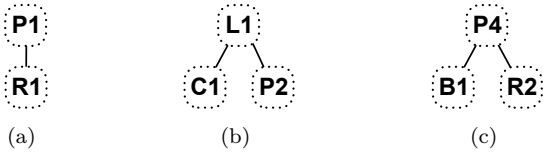


Figure 6.16. (a) The RPST of the process model in Figure 6.14, (b) the MDT and (c) the subtree of the RPST from the structuring scenario in Figure 6.15, all simplified

In the next section, we explain a technique for the maximal structuring of process models, which essentially boils down to the technique for synthesizing a process component from a given ordering relations graph.

6.2.2. Introduction to Maximal Acyclic Structuring

The open problem from Section 6.1 is to structure a given rigid process component into an equivalent maximally-structured process component R. In the light of Section 6.2.1, R has this property, iff (i) all primitive modules in the MDT of R's ordering relations graph are concurrent, and (ii) there exists a bijection between non-singleton modules of the MDT and non-simple components of the RPST which assigns to each primitive module a rigid component, to each complete a bond, and to each linear a polygon. The maximal structuredness of R follows from the maximality of the modular decomposition: the ordering relations graph of R inherits all information about well-structuredness from the proper prefix of R, and the MDT maximizes modules with a well-structured representation because of the decomposition into strong modules. If a concurrent primitive module M has well-structured child modules, then these modules are maximal again within M. Only the relations within M have no structured representation as a process model, where M is minimized by maximizing structuredness around and inside M. This yields a technique for maximal structuring: one must be able to synthesize a process component that exhibits the ordering relations described in M. Such a technique would allow defining unstructured process model topologies when mapping hierarchies of modules onto hierarchies of process components in Algorithm 6, e.g., the primitive module in Figure 6.15 onto a rigid component. The resulting process model would be maximally-structured.

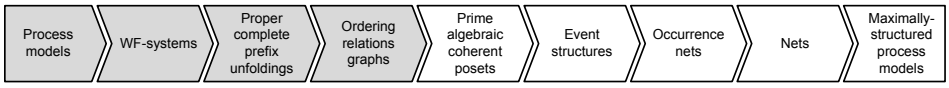


Figure 6.17. An extension of the structuring chain of Figure 6.1

In this section, we propose a solution to the synthesis problem, i.e., given an ordering relations graph (a module of an MDT) we synthesize a process model (a

component of the RPST) that realizes the relations described in the graph. The central idea is to first construct from the ordering relations graph an occurrence net that is interpreted as the unfolding of the process model and that exhibits the same ordering relations. *Refolding* this unfolding then yields the process model we wish to synthesize. The entire procedure requires several phases that employ results of domain- and net theory [96], and on folding prefixes of systems [40]. The procedure amounts to an extension of the structuring approach proposed in Figure 6.1 and is shown in Figure 6.17. The reconstruction of the occurrence net is explained in detail in Sections 6.2.3–6.2.5, and the refolding to a process model in Sections 6.2.6–6.2.8.

6.2.3. From Graphs to Partial Orders

This section describes a translation from an ordering relations graph into a partial order of information. The partial order is an alternative formalization of the behavior captured in the graph. The elements of the partial order essentially correspond to the configurations of a branching process (Definition 5.17).

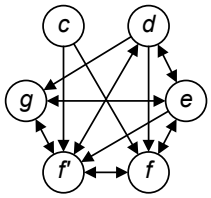


Figure 6.18. Ordering relations graph

The ordering relations graph in Figure 6.18 is a module from the running example of Section 6.2.1. The graph is a primitive module with all types of relations; f and f' represent events with the same label.

First, we give some definitions from the theory of partially ordered sets (posets) [96]. Let $(D, \sqsubseteq)$ be a poset. For a subset X of D, an element $y \in D$ is an upper (lower) bound of X, iff $x \sqsubseteq y$ ($x \sqsupseteq y$), for each element $x \in X$. An element $y \in D$ is a greatest (least) element, iff for each element $x \in D$ holds $x \sqsubseteq y$ ($x \sqsupseteq y$). An element $y \in D$ is a maximal (minimal) element, iff there exists no element $x \in D$, such that $y \sqsubset x$ ($x \sqsubset y$); D_{max} and D_{min} denote the sets of maximal and minimal elements of D. Two elements x and y in D are *consistent*, written $x \uparrow y$, iff they have a joint upper bound, i.e., $x \uparrow y \Leftrightarrow \exists\ z \in D : x \sqsubseteq z \wedge y \sqsubseteq z$; otherwise they are *inconsistent*. A subset X of D is *pairwise consistent*, written $X^{\Uparrow}$, iff every two elements in X are consistent in D, i.e., $X^{\Uparrow} \Leftrightarrow \forall x, y \in X : x \uparrow y$. The poset $(D, \sqsubseteq)$ is *coherent*, iff each pairwise consistent subset X of D has a least upper bound (lub) $\sqcup X$. An element $x \in D$ is a *complete prime*, iff for each subset X of D, which has a lub $\sqcup X$, holds that $x \sqsubseteq \sqcup X \Rightarrow \exists\ y \in X : x \sqsubseteq y$. Let $P = (D, \sqsubseteq)$ be a poset. We write $\mathfrak{P}_P$ for the set of complete primes of P. The poset $P = (D, \sqsubseteq)$ is *prime algebraic*, iff $\mathfrak{P}_P$ is denumerable and every element in D is the lub of the complete primes it dominates, i.e., $\forall\ x \in D : x = \sqcup\{y \mid y \in \mathfrak{P}_P \wedge y \sqsubseteq x\}$. A set S is *denumerable*, iff it

is empty or there exists an enumeration of S that is a surjective mapping from the set of positive integers onto S. Figure 6.19 shows two prime-algebraic posets; their elements are sets (of events) ordered by set inclusion. The complete primes are written in bold typeface.

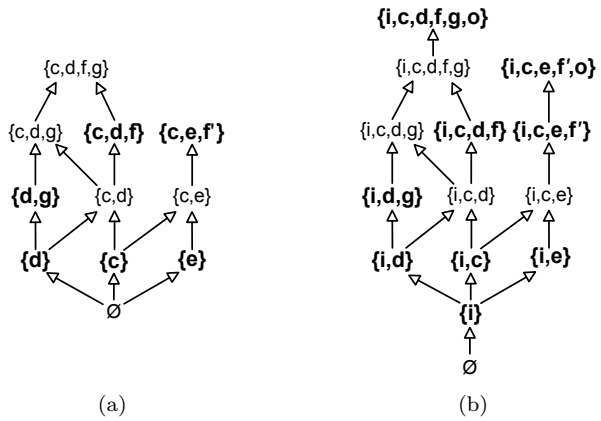


Figure 6.19. (a) Poset, and (b) augmented poset obtained from the graph in Figure 6.18

The behavior captured in an ordering relations graph can be given as a partial order of information points. Similar to [96], the information points are chosen to be left-closed and conflict-free subsets of vertices of the graph. In our case the graph's vertices represent events and each such set captures the history of events of some run of a system. Let $\mathcal{G} = (V, A, \mathcal{B}, \sigma)$ be a graph and let W be a subset of V. W is *conflict-free*, iff $\forall\ v_1, v_2 \in W : (v_1, v_2) \notin A \vee (v_2, v_1) \notin A$. W is *left-closed*, iff $\forall\ v_1 \in W\ \forall\ v_2 \in V : (v_2, v_1) \in A \wedge (v_1, v_2) \notin A \Rightarrow v_2 \in W$. We define $\mathcal{L}[\mathcal{G}]$ as the partial order of left-closed and conflict-free subsets of V, ordered by inclusion. Figure 6.19(a) shows the poset $\mathcal{L}$ of the graph in Figure 6.18. Theorem 6.16, inspired by Theorem 8 in [96], characterizes the posets $\mathcal{L}[\mathcal{G}]$.

Theorem 6.16. *Let $\mathcal{G} = (V, A, \mathcal{B}, \sigma)$ be an ordering relations graph, let $B = \{a \in A \mid a^{-1} \notin A\}$. Then, $\mathcal{L}[\mathcal{G}] = (H, \subseteq)$ is a prime algebraic coherent partial order. Its complete primes are those elements of the form $\lceil v \rceil = \{v' \in V \mid (v', v) \in B^*\}$.* ⋆

Proof. Let $X \subseteq H$ be pairwise consistent. Then, $\cup X$ is conflict-free. $\sqcup X = \cup X$ and, hence, $\mathcal{L}[\mathcal{G}]$ is coherent. Each $\lceil v \rceil$, $v \in V$, is clearly left-closed and conflict-free. Let $X \subseteq H$ have lub $\sqcup X$. X is pairwise consistent and $\sqcup X = \cup X$. Each $\lceil v \rceil$ is a complete prime. If $\lceil v \rceil \subseteq \cup X$, then $v \in \cup X$ and for some $x \in X$ holds $v \in x$ and, thus, $\lceil v \rceil \subseteq x$. It holds for each $X \in H$ that $X = \cup\{\lceil v \rceil \mid v \in \cup X\}$. Thus, each element of $\mathcal{L}[\mathcal{G}]$ is a lub of the complete primes below it. □

Given an ordering relations graph, one can construct $\mathcal{L}[\mathcal{G}] = (H, \subseteq)$ iteratively. Let h_1 and h_2 be subsets of V, such that $h_2 \setminus h_1 = \{v\}$. Then $h_1, h_2 \in H$, iff $h_1 = \varnothing$ or $\exists\ a \in h_1 : (v, a) \notin A$, and $\forall\ b \in h_1 : (b, v) \notin A \vee (v, b) \notin A$, and $\forall\ c \in V \setminus h_1, (c, v) \in A, (v, c) \notin A\ \exists\ d \in h_1 : (c, d), (d, c) \in A$.

Finally, we augment $\mathcal{L}[\mathcal{G}] = (H, \subseteq)$ with two fresh events $i, o \notin V$ to ensure the existence of a single source i and a single sink o in the synthesis result. An *augmented* partial order of $\mathcal{G}$ is $\mathcal{L}^*[\mathcal{G}] = (H^*, \subseteq)$, where $H^* = \{\varnothing\} \cup \{h \cup \{i\} \mid h \in H\} \cup \{h \cup \{i, o\} \mid h \in H_{max}\}$. Figure 6.19(b) shows $\mathcal{L}^*$ of the graph in Figure 6.18. Adding new minimal and maximal elements i and o leaves the topology of the posets unchanged, so $\mathcal{L}^*[\mathcal{G}]$ is a prime algebraic coherent poset.

6.2.4. From Partial Orders to Event Structures

The step from an ordering relations graph $\mathcal{G}$ to its augmented poset $\mathcal{L}^*[\mathcal{G}]$ basically describes configurations of an unfolding that has the same ordering relations as $\mathcal{G}$. The complete primes of $\mathcal{L}^*[\mathcal{G}]$ play a special role: they identify single events. Synthesizing the unfolding itself requires to define these events and their conflict relation explicitly. We do so with the help of the well-studied concept of an event structure [96].

Definition 6.17 (Labeled event structure).
An *event structure* is a triple $\mathcal{E} = (E, \leq, \oplus)$, where E is a set of events, $\leq$ is a partial order over E called the causality relation, and $\oplus$ is a symmetric and irreflexive relation in E, called the conflict relation that satisfies the principle of *conflict heredity*, i.e., $\forall e_1, e_2, e_3 \in E : e_1 \oplus e_2 \wedge e_2 \leq e_3 \Rightarrow e_1 \oplus e_3$. A *labeled* event structure $\mathcal{E} = (E, \leq, \oplus, \mathcal{C}, \kappa)$ additionally has a set $\mathcal{C}$ of *labels*, $\tau \in \mathcal{C}$, and $\kappa : E \to \mathcal{C}$ assigns to each event a label. ⌟

An ordering relations graph $\mathcal{G}$ differs from an event structure $\mathcal{E}$ in that $\mathcal{G}$ allows violations of conflict heredity. These violations, however, are not harmful; they express equivalent runs of a system. These equivalent runs are visible in the posets of Sect. 6.2.3 and become explicit in event structures. Each prime algebraic coherent poset $\mathcal{L}^*[\mathcal{G}] = (H, \subseteq)$ of an ordering relations graph $\mathcal{G}$ induces an event structure $\mathcal{E}[\mathcal{L}^*[\mathcal{G}]]$: each complete prime becomes an event, the ordering relation $\subseteq$ induces the partial order $\leq$ over events, conflicts arise between events that have no joint least upper bound in $\mathcal{L}^*[\mathcal{G}]$. The resulting event structure can intuitively be understood as an unfolding of the ordering relations graph that adheres to conflict heredity. The formal definition is an extension of Definition 18 in [96]; it incorporates propagation of labels of an originative ordering relations graph to the corresponding event structure.

Definition 6.18 (Event structure of partial order).
Let $\mathcal{G} = (V, A, \mathcal{B}, \sigma)$ be an ordering relations graph and let $P = (H, \subseteq)$ be an (augmented) prime algebraic coherent partial order of $\mathcal{G}$. Then, $\mathcal{E}[P]$ is defined as the labeled event structure $(E, \leq, \oplus, \mathcal{C}, \kappa)$, where $E = \mathfrak{P}_P$, $\leq$ is $\subseteq$ restricted to $\mathfrak{P}_P$, for all $e_1, e_2 \in \mathfrak{P}_P : e_1 \oplus e_2$, iff e_1 and e_2 are inconsistent in P, and $\mathcal{C} = \mathcal{B} \cup \{\tau\}$. Let $e \in E$, and define $\hat{e}$ as $\hat{e} \in e \setminus \bigcup_{a \subset e, a \in H} a$. Then, $\kappa(e) = \sigma(\hat{e})$, if $\hat{e} \in V$; otherwise $\kappa(e) = \tau$, for all $e \in E$. ⌟

Figure 6.20(a) visualizes $\mathcal{E}[\mathcal{L}^*[\mathcal{G}]]$ for the graph $\mathcal{G}$ of Figure 6.18. Events are complete primes of $\mathcal{L}^*[\mathcal{G}]$ (shown in boldface in Figure 6.19 and next to vertices in Figure 6.20). Directed edges encode causality (transitive dependencies are not

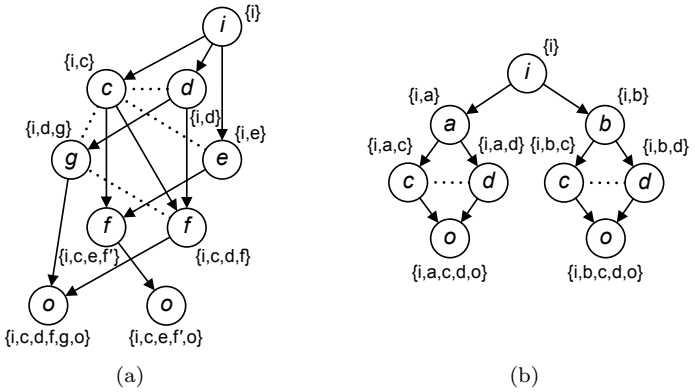


Figure 6.20. Event structures obtained from (a) Figure 6.19(b), and (b) the augmented poset of the orgraph in Figure 6.5(c)

shown), dotted edges represent implicit concurrency, whereas an absence of an edge hints at a conflict relation. The particular example of the event structure in Figure 6.20(a) is structurally similar to the graph in Figure 6.18; they differ only in relations with fresh i, o events. In general, event structures tend to have a different structure compared to the originative graphs. For instance, Figure 6.20(b) shows the event structure derived from the augmented poset of the orgraph in Figure 6.5(c).

6.2.5. From Event Structures to Occurrence Nets

The event structure $\mathcal{E} = \mathcal{E}[\mathcal{L}^*[\mathcal{G}]]$ explicitly describes events of an unfolding that has the same ordering relations as the originative ordering relations graph $\mathcal{G}$. We obtain the unfolding by enriching $\mathcal{E}$ with conditions, that is, we translate $\mathcal{E}$ to an occurrence net. Nielsen et al. in [96] show a tight connection between event structures and occurrence nets. Let $N = (B, E, G)$ be an occurrence net. Then, $\xi[N] = (E, G^* \cap E^2, \#_N \cap E^2)$ is a corresponding event structure. The next theorem, borrowed from [96], defines the converse: how to construct an occurrence net from a given event structure.

Theorem 6.19. *Let $\mathcal{E} = (E, \leq, \oplus)$, $E \neq \varnothing$, be an event structure. Then, there exists an occurrence net $\eta[\mathcal{E}]$, such that $\mathcal{E} = \xi[\eta[\mathcal{E}]]$.* ⋆

Proof. Define the set $CE = \{x \subseteq E \mid \forall e_1, e_2 \in x : e_1 \neq e_2 \Rightarrow e_1 \# e_2\}$. The events of $\eta[\mathcal{E}]$ are exactly those in E. The set of conditions is defined by $B = \{\langle e, x\rangle \mid e \in E, x \in CE, \text{ and } \forall e' \in x : e \leq e'\} \cup \{\langle 0, x\rangle \mid x \in CE, \text{ and } x \neq \varnothing\}$. The flow relation is defined by $G = \{(\langle e, x\rangle, e') \mid \langle e, x\rangle \in B, e' \in x\} \cup \{(\langle 0, x\rangle, e') \mid \langle 0, x\rangle \in B, e' \in x\} \cup \{(e, \langle e, x\rangle) \mid \langle e, x\rangle \in B\}$. It follows, that $\eta[\mathcal{E}] = (B, E, G)$ is an occurrence net for which $\# = \oplus$, and hence $\xi[\eta[\mathcal{E}]] = \mathcal{E}$. □

In the following we consider *labeled* event structures and correspondingly *labeled* occurrence nets to describe system behavior where a transition may occur in

several contexts, e.g., after a join. A labeled occurrence net generalizes the notion of a branching process of a (labeled) net system (Definition 5.15) as it describes system behavior when the system is *not* known – which is the case here, as we want to synthesize one.

A labeled occurrence net $N = (B, E, G, \mathcal{T}, \lambda)$ is an occurrence net (B, E, G) (Definition 5.12) where $\lambda : B \cup E \to \mathcal{T}$, $\tau \in \mathcal{T}$, assigns each node $x \in B \cup E$ a label $\lambda(x)$. Every branching process $\beta = (N, \nu)$, $N = (B, E, G)$ of a labeled net system $S = (N_S, M_0)$, $N_S = (P, T, F, \mathcal{T}, \lambda)$ induces the labeled occurrence net $(B, E, G, \mathcal{T}, \nu \circ \lambda)$ that describes the ordering of occurrences of labels $\mathcal{T}$ instead of occurrences of transitions of S.

Theorem 6.19 canonically lifts to labeled event structures and labeled occurrence nets: each labeled event structure $\mathcal{E} = (E, \leq, \oplus, \mathcal{C}, \kappa)$ induces the labeled occurrence net $\eta[\mathcal{E}] = (B, E, G, \mathcal{C} \cup \{\tau\}, \kappa')$ which preserves the labels of events and assigns each condition label τ, i.e., for all $e \in E, \kappa'(e) = \kappa(e)$, and for all $b \in B, \kappa'(b) = \tau$.

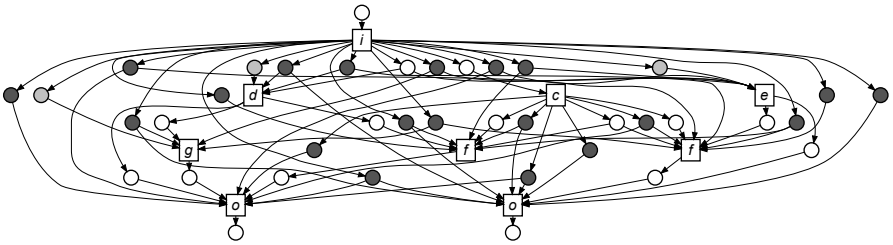


Figure 6.21. Occurrence net obtained from Figure 6.20(a) by Theorem 6.19 (without redundant conditions)

Figure 6.21 shows the labeled occurrence net which is constructed from the event structure shown in Figure 6.20(a) using the principles of Theorem 6.19. Theorem 6.19 defines a "maximal" construction, cf., [96], i.e., the resulting nets tend to contain much redundancy. With Definition 6.20 we aim at preserving only essential behavioral dependencies.

Definition 6.20 (Conditions). Let $N = (B, E, G, \mathcal{T}, \lambda)$ be a labeled occurrence net.

- A condition $b \in B$ is *redundant*, iff $b\bullet = \varnothing \wedge \exists\ b' \in B, b \neq b' : b' \in (\bullet b)\bullet$ or $\bullet b = \varnothing \wedge \exists\ b' \in B, b \neq b' : (b\bullet = b'\bullet) \wedge (\bullet b' \neq \varnothing)$.
- A condition $b \in B$ is *subsumed* by condition $b' \in B$, $b \neq b'$, iff $\bullet b = \bullet b' \wedge b\bullet \subseteq b'\bullet$.
- A condition $b \in B$ denotes a *transitive conflict* between events $e_1, e_2 \in E$, iff $\lambda(e_1) \neq \lambda(e_2)$ and there exists condition $b' \in B, b \neq b'$, and events $f_1, f_2 \in b'\bullet$, $(f_1 \rightsquigarrow_N e_1) \wedge (f_2 \rightsquigarrow_N e_2 \vee f_2 = e_2)$.
- Any other condition is *required*. ⌟

A redundant condition has no pre-event (post-event), and is not a pre-condition (post-condition) of the initial (a final) event. A subsumed condition b always has a sibling b' expressing the same constraints for larger set of events. A condition b denotes a transitive conflict between two events, if an "earlier" condition b' already

denotes this conflict. Subsumed conditions are shaded light-grey in Figure 6.21 and transitive conflicts dark-grey. All these conditions can be removed from the occurrence net without loosing information about ordering of events.

An exception to transitive conflicts is a condition b shared by two post-events e_1, e_2 with the same label. Here, b not only expresses conflict, but also the *only* direct causal dependency of e_1 and e_2 on the pre-event of b; such a condition is required (Definition 6.20).

For synthesizing a Petri net from the occurrence net obtained from an ordering relations graph, we first emove from the occurrence net all redundant conditions, then all subsumed conditions, and finally all transitive conflicts. Removing these conditions from the net in Figure 6.21 yields the net in Figure 6.24. Note that all conditions are labeled τ. Condition b_9 highlights the exception to transitive conflicts: it is the only condition expressing that f depends on c and, hence, must be part of the occurrence net. Also the resulting net still contains a number of *implicit conditions*, i.e., conditions which could be removed from the net without changing the ordering relations such as b_{11} denoting $e_2 \rightsquigarrow e_8$, which is also expressed by the path $e_2, b_8, e_6, b_{15}, e_8$. We shall see that these remaining implicit conditions are vital for synthesizing a Petri net from a given occurrence net.

6.2.6. From Occurrence Nets to Nets – The Basic Idea

After removing redundant, subsumed, and transitive conflict conditions (Definition 6.20), the occurrence net obtained by Theorem 6.19 is already a process model – although one with duplicate structures and multiple sinks. We obtain a more compact model with a single sink by *folding* the occurrence net.

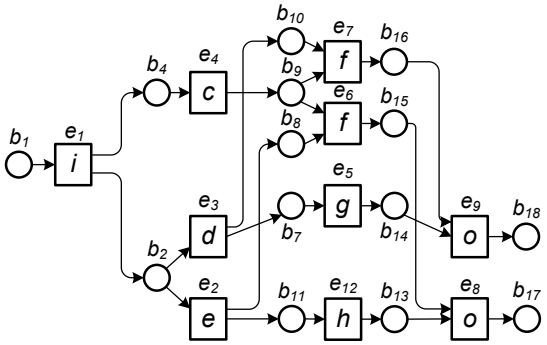


Figure 6.22. Occurrence net without implicit conditions

We first illustrate the idea on a simple, special case and then present the general approach. Consider the occurrence net N in Figure 6.22, which contains *no* implicit conditions. Intuitively, we obtain a process model from N by folding any two nodes of N which have isomorphic successors into one node. This operation preserves all ordering relations and all behavior represented in N. The corresponding formal notion is an equivalence on the nodes of the occurrence net, called *future equivalence*, which is characterized co-inductively:

Two nodes of an occurrence net are future equivalent only if they have the same label and their postsets are future equivalent.

The equivalence classes of the future equivalence define the nodes of the folded net: for each equivalence class, fold all its nodes into one node, preserving the arcs. The branching process of the folded net is exactly the original occurrence net, see [40, Theorem 8.7].

For the occurrence net N in Figure 6.22, consider the future equivalence $\sim_f$ with the classes $\{b_{17}, b_{18}\}$, $\{e_8, e_9\}$, $\{b_{15}, b_{16}\}$, $\{b_{13}, b_{14}\}$, $\{e_6, e_7\}$, $\{b_8, b_{10}\}$, and all other nodes remaining singleton. Folding N under $\sim_f$ yields the net in Figure 6.23 which has N as its branching process.

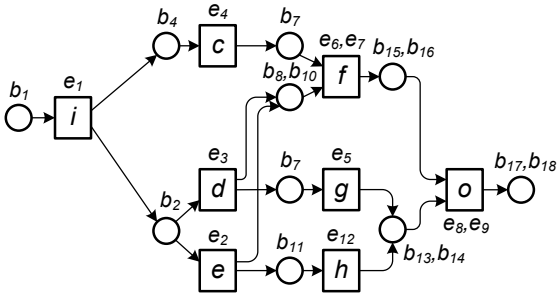


Figure 6.23. Folded net obtained from Figure 6.22

Each occurrence net has several future equivalences differing in how pre-conditions of events are folded. The algorithm for constructing a future equivalence is described next.

6.2.7. From Occurrence Nets to Nets – The General Case

The general case of folding an occurrence net into a process model has one additional twist: the process model to be synthesized from the original ordering relations graph may require *unobservable* control flows between gateways *without* a task, e.g., the flow from x to y in Figure 6.14 is unobservable. Such flows can only be synthesized when the occurrence net to fold contains implicit conditions. The occurrence net obtained from Theorem 6.19 and reduced to its required conditions contains all implicit conditions having exactly one pre- and one post-event. These implicit conditions are an overapproximation of the unobservable control flows in the process model. During folding, our folding procedure identifies all implicit conditions that explain required unobservable control flow and discards all others. Technically, we fold an occurrence net by first identifying a *folding equivalence* and then folding all nodes of an equivalence class into the same node of the process model. The folding equivalence has two properties: (i) it is a future equivalence, so that equivalent events of an occurrence net have equivalent successor events, and (ii) pre-conditions and post-conditions of equivalent events have to be mutually equivalent. We first formalize these notions, and then describe the procedure that finds such a folding equivalence for a given occurrence net.

Definition 6.21 (Future equivalence).
Let $N = (B, E, G, \mathcal{T}, \lambda)$ be a labeled occurrence net. An event $e \in E$ is a *direct successor* of an event $e' \in E$, written $e \in dSucc(e')$, iff $e' \rightsquigarrow_N e$ and for no $e'' \in E$ holds $e' \rightsquigarrow_N e'' \rightsquigarrow_N e$.

An equivalence relation $\sim \subseteq (B \times B) \cup (E \times E)$ is a *future equivalence* on N, iff the following properties hold:

- For all $x, y \in B \cup E$ holds, $x \sim y$ implies $\lambda(x) = \lambda(y)$ and $\neg(x \parallel_N y)$.
- For all $e, f \in E$ with $e \sim f$ holds: for each $e' \in dSucc(e)$ with $\lambda(e) = a$ exists $f' \in dSucc(f)$ with $\lambda(f) = a$ and $e' \sim f'$, and vice versa. ⌟

The future equivalence captures the essence of our behavioral equivalence criterion for structuring, fully concurrent bisimulation (Definition 5.11), in a stronger form that suits folding. Every future equivalence on an occurrence net N yields a fully concurrent bisimulation relation on the partially ordered runs described by N.[1] The converse does not hold as a future equivalence also considers invisible events of N. Merging future equivalent events of N into the same transition preserves the ordering relations encoded in N, see [40, Theorem 8.7] for the formal proof.

Folding an occurrence net N also requires to correctly fold pre- and post-conditions of events while treating implicit conditions in the right way. Thereby, the event with the largest number of non-implicit pre-conditions in an equivalence class determines the number of predecessors for the entire class; all other events in the class "fill up" their preset with implicit conditions, correspondingly for postsets. To formalize this, we need to introduce some notions.

For an occurrence net $N = (B, E, G, \mathcal{T}, \lambda)$, let $implicit_N \subseteq B$ be the set of implicit conditions of N, i.e., $b \in implicit_N$ iff $\{e_1\} = {\bullet b}, \{e_2\} = b{\bullet}$ and there is a path from e_1 to e_2 in N without b.

Let $\sim \subseteq (B \times B) \cup (E \times E)$ be an equivalence relation. For an equivalence class $\langle x \rangle$ of $\sim$, let $maxpre(\langle x \rangle)$ be the largest number of non-implicit predecessors of all members of $\langle x \rangle$, i.e., $maxpre(\langle x \rangle) = |{\bullet x'}|, x' \in \langle x \rangle$ s.t. for all $x'' \in \langle x \rangle$ holds $|{\bullet x'} \setminus implicit_N| \geq |{\bullet x''} \setminus implicit_N|$; correspondingly let $maxpost(\langle x \rangle)$ be the largest number of non-implicit successors of all members of $\langle x \rangle$. For sets $X, Y \subseteq B$ we write $X \sim Y$ iff $X = \{x_1, \ldots, x_k\}, Y = \{y_1, \ldots, y_k\}$ and $x_i \sim y_i$, for all $1 \leq i \leq k$.

Definition 6.22 (Folding equivalence).
Let $N = (B, E, G, \mathcal{T}, \lambda)$ be a labeled occurrence net, $\forall\ b \in B : \lambda(b) = \tau$. An equivalence relation $\sim \subseteq (B \times B) \cup (E \times E)$ *preserves the environment of events* iff for each equivalence class $\langle e \rangle$ of $\sim$, $e \in E$, and for all $e_1, e_2 \in \langle e \rangle$ holds

- $\exists B_1 \subseteq {\bullet e_1}, B_2 \subseteq {\bullet e_2} : B_1 \sim B_2 \wedge \forall i \in \{1, 2\} : |B_i| = maxpre(\langle e \rangle) \wedge {\bullet e_i} \setminus B_i \subseteq implicit_N$, and
- $\exists B_1 \subseteq e_1{\bullet}, B_2 \subseteq e_2{\bullet} : B_1 \sim B_2 \wedge \forall i \in \{1, 2\} : |B_i| = maxpost(\langle e \rangle) \wedge e_i{\bullet} \setminus B_i \subseteq implicit_N$.

An equivalence $\sim_f \subseteq (B \times B) \cup (E \times E)$ is a *folding equivalence* on N iff $\sim_f$ is a future equivalence and preserves the environment of events. ⌟

[1] Every prefix π of a labeled occurrence net N that contains no conflicting events is a partially ordered run [34].

Considering the occurrence net N in Figure 6.24, the equivalence $\sim_f$ with the classes $\{b_{17}, b_{18}\}$, $\{e_8, e_9\}$, $\{b_{11}, b_{14}\}$, $\{b_{15}, b_{16}\}$, $\{e_6, e_7\}$, $\{b_8, b_{10}\}$, and all other nodes remaining singleton, is a folding equivalence on N. In particular, in the equivalence class $\{e_8, e_9\}$ the preset of event e_9 defines that all events have to have 2 pre-conditions that are mutually equivalent. Hence, $\{b_{15}, b_{16}\}$ and $\{b_{11}, b_{14}\}$ are equivalence classes where the second class contains the implicit condition b_{14}. The remaining pre-conditions of $\{e_8, e_9\}$ need not be equivalent.

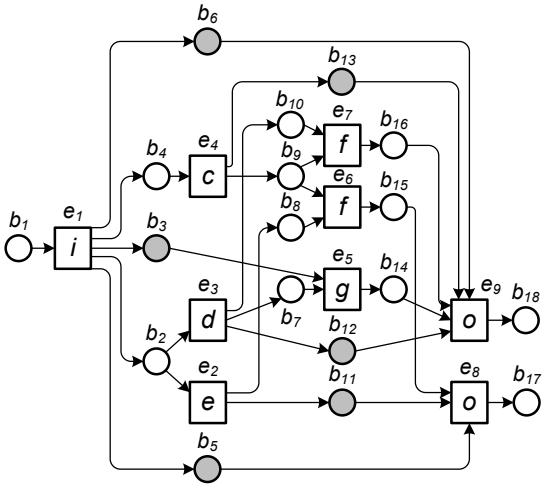


Figure 6.24. Simplified occurrence net obtained from Figure 6.21

Without implicit conditions, folding would not succeed, as it can be seen from the occurrence net in Figure 6.24, where implicit conditions are highlighted grey. The two equivalent events e_8 and e_9 differ in the number of non-implicit pre-conditions. To consistently fold e_8 and e_9 w.r.t. their pre- and postsets, the preset of e_8 has to be extended with one implicit condition during folding.

We fold an occurrence net to a Petri net by merging all nodes of a folding equivalence class into the same node – except for equivalence classes consisting of implicit conditions only, these are discarded.

Definition 6.23 (Folded net).
Let $N = (B, E, G, \mathcal{T}, \lambda)$ be a labeled occurrence net such that $\forall\ b \in B : \lambda(b) = \tau$. Let $\sim_f$ be a folding equivalence on N; write $\langle x \rangle_f = \{y \mid y \sim_f x\}$ for the equivalence class of x. Then, the *folded net of N under $\sim_f$* is the net $N_f = (B_f, E_f, G_f, \mathcal{T}, \lambda_f)$ where

- $B_f = \{\langle b \rangle_f \mid b \in B, \langle b \rangle_f \notin \mathit{implicit}_N\}$,
- $E_f = \{\langle e \rangle_f \mid e \in E\}$,
- $G_f = \{(\langle x \rangle_f, \langle y \rangle_f) \mid (x, y) \in G, \langle x \rangle_f, \langle y \rangle_f \in B_f \cup E_f\}$, and
- $\lambda_f(\langle x \rangle_f) = \lambda(x)$.

Folding N under $\sim_f$ described above yields the net N_f shown in Figure 6.25. Folding N into N_f preserves the behavior of N, see [40, Theorem 8.7].

A simple algorithm for computing a folding equivalence traverses the given finite occurrence net backwards in a breadth-first manner. The conditions of the occurrence net without any post-event have equivalent futures. Correspondingly, their pre-events with the same label have equivalent futures. Building the folding equivalence backwards in this way ensures that only future equivalent events are put into the same equivalence class. Branching and backtracking are used whenever for a condition b there are two or more pairwise concurrent conditions that could be folded with b. Each option is explored and the most-compact folding is chosen. For instance, in Figure 6.22 after folding $b_{17} \sim_f b_{18}$ and $e_8 \sim_f e_9$, for b_{15} the folding options b_{14} and b_{16} can be explored; backtracking yields b_{16} as the better match for b_{15} because of their f-labeled pre-events.

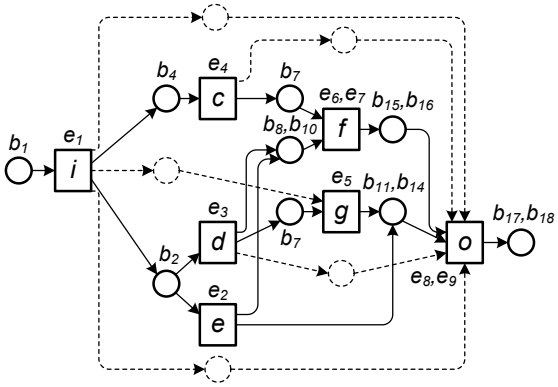


Figure 6.25. Folded net obtained from Figure 6.24

When extending the folding equivalence to pre-conditions of equivalent events E', for instance for $E' = \{e_8, e_9\}$, implicit conditions are taken into account as follows:

- Pick the event $e \in E'$ with the largest set B' of non-implicit pre-conditions, e.g., $\{b_{14}, b_{16}\} \subset {}^\bullet e_9$.
- For each $b \in B'$, extend the folding equivalence with a non-implicit or implicit condition $b' \in {}^\bullet e'$, for all other events $e' \in E'$, preferring non-implicit conditions over implicit ones, e.g., $b_{16} \sim_f b_{15}$, $b_{14} \sim_f b_{11}$.
- Finally, remove all non-implicit conditions not required in this step, e.g., b_5, b_6, b_{13}.

The second step ensures that pre-sets of equivalent events are preserved. The third step ensures that post-sets of equivalent events (that are identified in subsequent steps) are preserved. Various heuristics improve exploration and backtracking when matching pre-conditions of events with each other. When the future equivalence cannot be extended further, the net can be folded according to Definition 6.23. Applying this procedure on our example yields the folded net shown in Figure 6.25 without the dashed conditions and arcs.

6.2.8. From Nets to Process Models

The folding was the second to last step in synthesizing a process model from a given ordering relations graph. We obtained a Petri net N_f which we now transform into a process model P using the principles of Folded net 2.24 in the reverse direction.

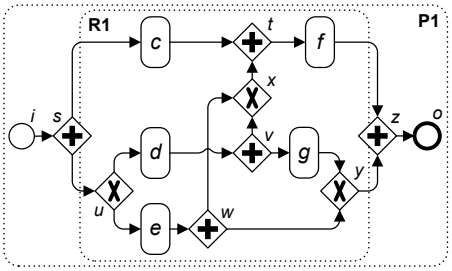


Figure 6.26. Process model obtained from Figure 6.25

The initial transition i (final transition o) is mapped to the source (sink) node of P. Every other transition of N_f becomes a task of P. Gateways of P follow from non-singleton pre- and postsets of nodes of N_f. A transition t with two or more pre-places is preceded by an *and* join; two or more post-places of t define an *and* split; the pre- and postsets of places define *xor* splits and joins, respectively; *and* gateways are always positioned closer to the task. In our example, $e_1\bullet$ defines *and* split s in Figure 6.26, $b_2\bullet$ defines *xor* split u, $e_3\bullet$ defines *and* split v, $\bullet\langle e_6, e_7\rangle$ defines *and* join t, and $\bullet\langle b_6, b_8\rangle$ defines *xor* join x positioned between t and v (*and* gateways closer to tasks); correspondingly for all other gateways. The arc from e_2 to $\langle b_{11}, b_{14}\rangle$ which was obtained from a transitive conflict (Definition 6.20) results in an important control-flow arc from w to y without any task.

This concludes our technique for maximal structuring. Returning to our running example, we complete the maximal structuring of the process model in Figure 6.14 by placing the synthesized process component $R1$ in Figure 6.26 at the corresponding spot in the RPST in Figure 6.15(bottom). Recall that this RPST was obtained from the unstructured process model using Algorithm 6. Placing $R1$ at its designated spot yields the maximally-structured model shown in Figure 6.27.

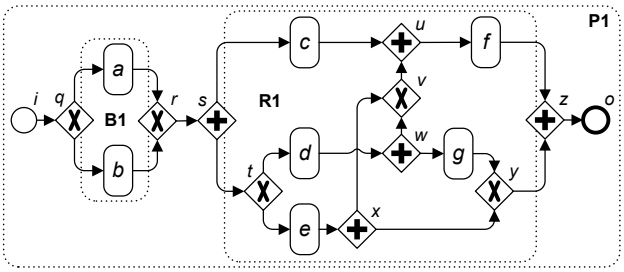


Figure 6.27. Maximally-structured version of the process model in Figure 6.14

6.2.9. Evaluation

The overall approach has been implemented in a tool named `bpstruct`, which is publicly available[2]. Using `bpstruct` we conducted an evaluation to assess the performance of our techniques and to analyze the amount of duplication introduced during the structuring. All tests were performed on a laptop with a dual core Intel processor, 2.53 GHz, 4 GB of memory, running Microsoft Vista and SUN Java Virtual Machine version 1.6 (with 512 MB of allocated memory). To eliminate load time from the measures, each test was executed five times, and we recorded the average execution time of the second to fifth run. In the following, we provide details about the dataset used for the study and discuss the results of the experiments.

Dataset

The study was conducted on a collection of process models extracted from industrial practice that has been publicly released for research purposes [42]. More precisely, we used the set of WF-nets provided in the `Woflan` [151, 150] file format[3]. In contrast to the original collection, where a large number of process models has multiple start nodes and/or multiple end nodes, the WF-nets have been completed such that each WF-net has a single source and a single sink place while preserving the original behavior. The reader is referred to [42] for a detailed description of the dataset and the underlying completion process.

	Structurable	**Maximally structurable**
Number of models	110	5
Avg number of nodes/arcs	21/31 (14/16)	32/51 (40/47)
Max number of nodes/arcs	119/195 (1178/1346)	124/173 (545/631)

Table 6.1. Structural information on structurable process models in the dataset

In a first stage we removed all unsound WF-nets from the collection. Every sound WF-net was parsed using the RPST decomposition as described in Chapter 3 and Chapter 7. Every bond and polygon in the RPST was abstracted into a single transition. For each rigid component identified in the RPST, a process model was synthesized using the approach described in Section 6.2.8. Two nodes were added to mark the entry and exit points, respectively. After this step, we ended with a total of 170 sound unstructured process models. Among them, 115 are *heterogeneous* acyclic rigids (acyclic rigids with *xor* and *and* gateways), 39 are *and* rigids (acyclic rigids with *and* gateways only), 14 are cyclic rigids, and 2 are *xor* rigids (rigids with *xor* gateways only). For the purpose of this study, we only kept models with heterogeneous acylic rigids and *and* rigids (154 process models). To ease the analysis, we further classified the heterogeneous rigids into "structurable" and "maximally structurable", accordingly. Table 6.1 and Table 6.2 summarize the size of the models used for the study. In Table 6.2, *and* rigids are referred to

[2] `http://code.google.com/p/bpstruct`
[3] `http://www.informatik.uni-rostock.de/~nl/wiki/soundness`

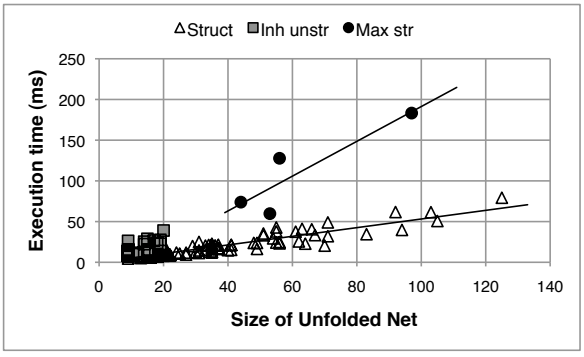
as *inherently unstructured*. The size of the models after structuring is shown in parenthesis. There were two exceptionally large process models in the dataset, one "structurable" and the other one "maximally structurable". Both models have been excluded when calculating the average sizes. However, their sizes correspond to maximum values for their structural classes in Table 6.1.

	Inherently unstructured
Number of models	39
Avg number of nodes/arcs	12/14
Max number of nodes/arcs	26/32

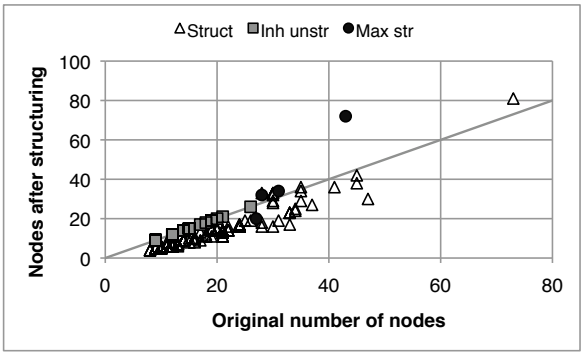
Table 6.2. Structural information on inherently unstructured process models in the dataset

Results

Our study reveals that out of the 154 process models, 110 models (71.43%) can be structured into well-structured models. More importantly, there exist models, in our case 5 (3.25%), which require the extension proposed in Section 6.2 to achieve maximal-structuring, while 39 models (25.32%) are inherently unstructured.



(a) Average structuring time



(b) Ratio of duplication

Figure 6.28. Experimental results

Concerning performance, Figure 6.28(a) presents the execution times relative to the number of events in the proper complete prefix unfolding, including a trend line for each structural class. It can be easily noted that the execution time is highly correlated to the size of the proper prefix. We observed a significant difference in the processing time for "maximally structurable" models compared to "well-structurable" models. Nevertheless, the average time for structuring was in the order of milliseconds, with a few exceptions. We found two exceptionally large cases: one model ("structurable") with 119 nodes (the proper prefix had 2429 events) and the other one ("maximally-structurable") with 124 nodes (the proper prefix had 913 events), requiring 9 and 1 seconds for structuring, respectively.

Figure 6.28(b) presents the variation in the size of models, i.e., number of nodes, after structuring. On average, the model's size decreased by about 15%, with a standard deviation of 0.817. This can be easily confirmed in the figure, as most of the points are located under the diagonal (which corresponds to a ratio 1:1), particularly in the case of "structurable" models. Recall that models were, in most of the cases, augmented with some additional elements (i.e., *and* gateways and transitive flow relations) to transform them into single exit models. Therefore, the reduction in size can be explained by the removal of gateways and transitive flow during the structuring performed by `bpstruct`. Conversely, the sizes of *and* rigids remained the same for this dataset. In additional internal experiments, we found that for certain *and* rigids it is also possible to observe a reduction on the size of the (maximally) structured version when redundant elements are eliminated, e.g., spurious flow relations/gateways. The dataset used in our experiments is publicly available at the `bpstruct` web site[4], including the graphical version of the process models in PDF file format.

6.3. Multi-Source and/or Multi-Sink Acyclic Structuring

In Section 6.1 and Section 6.2, we assumed that process models have single source and single sink nodes. However, contemporary notations for business process modeling, including BPMN and EPC, support the definition of models with multiple source and/or multiple sink nodes, hereafter called multi-source and multi-sink models. In this section, we extend the notion of structuredness for multi-source and multi-sink models and we extend the structuring method accordingly.

6.3.1. Notion of Structuredness

In the context of single-source and single-sink process models, we have defined a well-structured process model as one whose RPST does not contain any rigid components, see Section 5.1.1. This notion needs to be extended for the case of multi-source and multi-sink models. To this end, a process model with multiple source nodes is augmented with an additional (start) node and an arc from this fresh node to each of the source nodes. This additional node is labeled s by

[4] http://code.google.com/p/bpstruct/wiki/MaxStructEvaluation

convention. Conversely, a model with multiple sink nodes is augmented with an additional (end) node and an arc from each of the sink nodes of the original model to this fresh node (labeled e by convention). This augmentation is captured by the following definition.

Definition 6.24 (Augmented process model).
Let PM be a multi-source and multi-sink process model. The *augmented* version of PM is constructed from PM as follows:

- If PM has more than one source, a new source *start* is added and for each source node s of PM, an arc from *start* to s is added.
- If PM has more than one sink, a new sink *end* is added and for each sink node s of PM, an arc from s to *end* is added. ⌟

A multi-source and multi-sink model is said to be (well-)structured, iff the RPST of its augmented version contains no rigid process component. Please note that the augmentation of a model aligns well with the principles of the generalized RPST computation, see Section 3.4.2 for details.

Given an unstructured multi-source and multi-sink process model PM, our goal is to compute an FCB-equivalent structured process model PM'. By definition, this means that the labels of the nodes in PM' must coincide with those in PM. Hence, the special nodes s and e, which are added for the sake of constructing the RPST, need to be removed at the end of the structuring procedure, thereby yielding a structured multi-source and multi-sink process model as output.

6.3.2. Instantiation Semantics

Given a multi-source and multi-sink process model, we can compute the RPST of the augmented model in order to separate rigid components from non-rigid ones. Non-rigid components are already structured and thus can be replaced with a single "black box", so that their parent node in the RPST can be structured. For example, Figure 6.29(a) shows a multi-source and multi-sink model, captured in EPC notation. Figure 6.29(b) shows its augmented version, whose RPST contains rigid component $R1$, under which we can find two bond components $B1$ and $B2$. Each of these bond components in turn contains two polygons, but the polygons are not shown in the figure for the sake of simplicity. After replacing components $B1$ and $B2$ with black-boxes, we obtain the abstract model depicted in Figure 6.29(c) consisting of a single rigid component. The problem of structuring the original EPC is then reduced to that of structuring this rigid component.

Rigid components can be classified into those that contain one of the special nodes s or e and those that do not. The latter type of rigid can be structured using the method outlined in the previous sections. Therefore, we can focus our attention on the case where a rigid contains s or e. Without loss of generality, we assume below that we are given a rigid component that contains both the s node and the e node introduced during the augmentation. The case where a rigid contains either the s node or the e node (but not both) is just a special case.

The first step in structuring a rigid component is to compute its corresponding net. Definition 2.24 can be directly applied to multi-sink process models (and thus

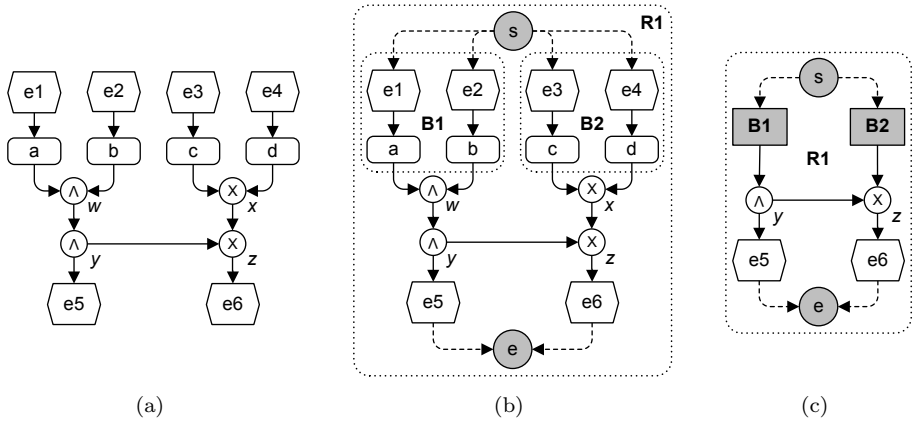


Figure 6.29. (a) A multi-source and multi-sink EPC, (b) its augmented version, and (c) its abstract version

to multi-sink rigid components). The resulting net will have multiple sink places, but this feature does not pose any particular problem. Indeed, the unfolding is defined for multi-sink nets in the same way as it is for single-sink nets. On the other hand, the mapping of multi-source process models to Petri nets requires special care for two reasons:

1. In order to compute an unfolding, we need to define an "initial marking". In the case of a single-source net, the initial marking is the one that contains a single token in the source place and no other tokens, but in the case of multi-source nets, several initial markings are possible.
2. Different process modeling notations adopt different semantics for multi-source models [21].

Hence, we need to consider each process modeling notation separately in order to determine how to map a multi-source process model (or a process component) in that notation into a Petri net, and how to determine the initial marking from which the unfolding will be computed. Below we address these questions in the context of two concrete process modeling notations, namely BPMN and EPC.

Multi-source BPMN Models

The notion of a process model (as per Definition 2.23) allows us to generically represent models in several graph-oriented process modeling notations, including BPMN and EPC. Below, we consider the case where a process model represents a BPMN model. In this context, a node in a process model represents either a BPMN activity, a BPMN event, or a BPMN gateway. Such process models may contain multiple source nodes, each one corresponding either to a "start event" or to an "event-based gateway". As per the BPMN standard specification [2], the instantiation semantics of such multi-source models is as follows:

- If a BPMN model starts with multiple events, a new process instance is created whenever one of these "start events" fires. The mapping of this case to a Petri net with a single start place is depicted in Figure 6.30(a).
- If a BPMN model starts with multiple event-based gateways that participate in a common conversation, each of these gateways must receive one token. The corresponding Petri net mapping is shown in Figure 6.30(b).
- If a BPMN model starts with a so-called "parallel event-based gateway", then each one of the events connected to this parallel event-driven gateway must occur before the process is instantiated. The corresponding Petri net mapping is shown in Figure 6.30(c).

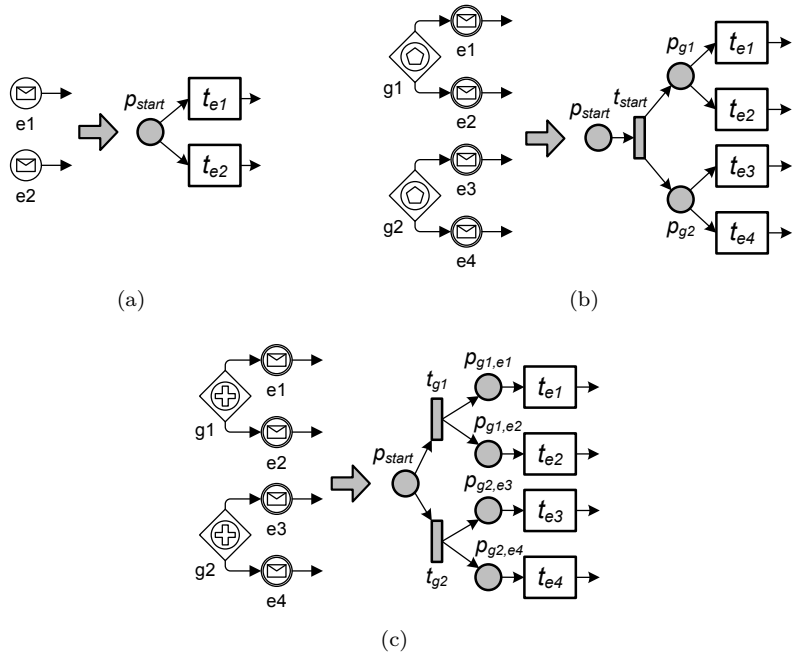


Figure 6.30. Mapping multi-source BPMN models to nets

In all three cases, we see that a multi-source BPMN model – and consequently a rigid component in a BPMN model containing the s node – can be mapped to a Petri net with a single source place such that the initial marking consists of exactly one token in this source place. Since this mapping is trivial, as shown in Figure 6.30, we omit its formal definition. The rest of the section focuses on the EPC semantics, which requires a more extensive treatment.

Multi-source EPC Models

There is no "official" precise instantiation semantics for EPCs with multiple start events. However, some authors have adopted the following semantics [21]:

- An instance of the process requires at least one of the start events to be triggered.
- Additional start events may be triggered during the execution of a process instance.

An EPC can be represented as a process model (as per Definition 2.23) where each node of the process model represents either a function, an event, or a connector. In order to capture the instantiation semantics of an EPC process model with multiple source tasks, the nets obtained by applying Definition 2.24 can be augmented to a single source.

Definition 6.25 (Augmented Petri net).
Let $N = (P, T, F)$ be a net and let S be the set of all source places of N. The *augmented* version of N is constructed from N as follows:

- A fresh place p_{start} is added in the net.
- For every non-empty subset of source places $\mathbf{s} \in \mathcal{P}(S) \setminus \emptyset$, a fresh *start* transition $t_{\mathbf{s}}$ and a fresh flow arc $(p_{start}, t_{\mathbf{s}})$ is added in the net.
- For every source place $s \in S$ and for every non-empty subset of source places $\mathbf{s} \in \mathcal{P}(S) \setminus \emptyset$ containing s, i.e., $s \in \mathbf{s}$, a fresh flow arc $(t_{\mathbf{s}}, s)$ is added in the net.

For example, Figure 6.31 shows the augmented version of the net which corresponds to the abstract EPC model in Figure 6.29(c) (note the abuse of notation for simplicity reasons). Fresh nodes are highlighted with grey background. Place p_{start} is the only source place of the resulting net.

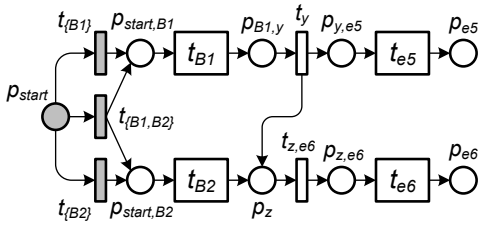


Figure 6.31. The augmented version of the net that corresponds to the abstract model in Figure 6.29(c)

If we set the initial marking to be the one that contains a single token in the source place (and no tokens elsewhere), the augmented net captures all possible *instantiations* of the process model. The Petri net of a multi-source and multi-sink process model (or component of a model) starts with one transition per possible instantiation. For instance, the net in Figure 6.31 starts with three transitions: transition $t_{\{B1\}}$ corresponds to the instantiation where only component $B1$ is executed, transition $t_{\{B2\}}$ corresponds to the instantiation where only component $B2$ is executed, and transition $t_{\{B1,B2\}}$ corresponds to the case where both components are executed.

6.3.3. Soundness

An instantiation of a process model does not always lead to a successful completion of the process. Some instantiations may lead to *deadlocks* or *lack of synchronization*. Here, a deadlock is defined as a situation where no transition can fire, but one of the branches is still active. In other words, a deadlock occurs when there is a token in a non-sink place in the net, and no transition in the net is enabled. Lack of synchronization refers to a situation in which a transition can fire twice (without any other transition firing in-between these two firings). This corresponds to the situation where the net reaches a marking where there is more than one token in a place. Deadlock freeness and proper synchronization, i.e., absence of any state exhibiting a lack of synchronization, correspond to the notions of soundness and safeness introduced earlier in this article [147].

In light of the above, Gero Decker and Jan Mendling [21] suggest – but do not formally define – a notion of "correct instantiation" of an EPC based on the idea that a start event will be triggered, if and only if it is required, meaning that:

- If the execution of a process instance runs into a deadlock because one of its join gateways is waiting for one of its branches to complete, and the completion of this branch requires one of the start events to be triggered, this start event will eventually be triggered so that the execution of the process instance can complete.
- A start event will not be triggered if this may eventually cause a *lack of synchronization*.

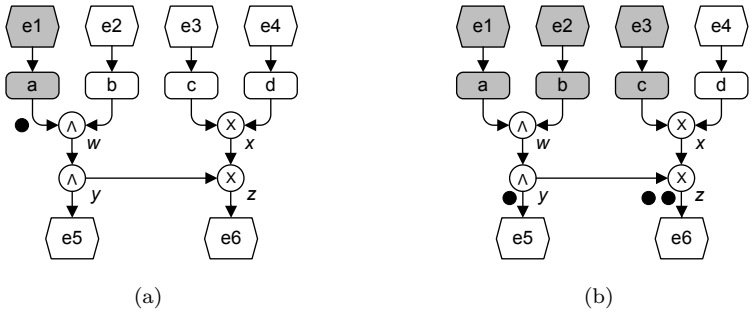


Figure 6.32. Markings of EPCs with (a) "event must occur" situation, and (b) lack of synchronization

In order to illustrate the notions of deadlock and lack of synchronization, let us go back to the EPC in Figure 6.29(a). The execution of the process may start with an occurrence of event $e1$. Eventually, the left-hand side branch of gateway w completes, i.e., a token reaches the left-hand side incoming arc of the gateway. This is the state depicted in Figure 6.32(a). In order to proceed, a token is required in the second incoming branch. Otherwise, the execution would remain deadlocked in gateway w. Thus, event $e2$ must eventually occur. On the other hand, we note that neither event $e3$ nor event $e4$ may occur at this stage, since that would lead

to a lack of synchronization, depicted in Figure 6.32(b), which shows a state where event $e6$ may occur twice.

Coming back to the abstract model in Figure 6.29(c), we observe that the instantiation where either $B1$ or $B2$ are executed is correct. On the other hand, the instantiation where both $B1$ and $B2$ are executed leads to the lack of synchronization depicted in Figure 6.32(b). In other words, start transition $t_{\{B1,B2\}}$ in Figure 6.31 corresponds to an incorrect instantiation of the original model. We note in passing that in the abstract EPC, the deadlock situation depicted in Figure 6.32(a) does not manifest itself because the underlying events have been abstracted away inside $B1$.

In order to capture the notion of "correct instantiation" introduced in [21], we proceed as follows: We start with the augmented version of a net that corresponds to a multi-source process model, as per Definition 6.25. Based on this net, we compute an unfolding starting from the initial marking that puts one token in the source place and no tokens elsewhere. This unfolding captures both the correct and incorrect instantiations. Accordingly, we "prune" the unfolding in order to remove those branches that represent incorrect instantiations. To this end, we formally capture – at the level of the unfolding – the notion of incorrect instantiation, i.e., lack of synchronization and deadlock. The following definition – based on similar definitions by Dirk Fahland [40] – captures these notions. Here, the term "locally unsafe condition" is used in lieu of "lack of synchronization".

Definition 6.26 (Local safeness, Local deadlock).
Let $\beta = (N, \nu)$, $N = (B, E, G)$, be the unfolding of an acyclic system $S = (N', M_0)$.

- A condition $b \in B$ is *locally safe* in β, iff there exists no condition $c \in B$, $b \neq c$, such that b is concurrent to c and both correspond to the same place in N', i.e., $\nexists\, c \in B, b \neq c : (b \parallel_N c) \wedge (\nu(b) = \nu(c))$; otherwise b is *locally unsafe*.
- A condition $b \in B$ is a *local deadlock* in β, iff one of the following holds:
  - $b\bullet = \varnothing$ and $\nu(b)\bullet \neq \varnothing$.
  - There exist event $e \in b\bullet$, condition $c \in \bullet e$, and condition $d \in B$, such that: $d\bullet = \varnothing$, $d \#_N b$, and $d \parallel_N c$. ⌟

A locally unsafe condition in a branching process of a net system clearly signals that the system is unsafe, see Proposition 6.3 in [38]. Every local deadlock hints at the existence of a deadlock in the system. Definition 6.26 specifies the deadlock condition in the special case of acyclic systems. In case of the general class of systems, one can deduce deadlock conditions by following the principles described in [83, 85]. Accordingly, the *sound* unfolding of a system is a prefix of its unfolding that excludes incorrect instantiations.

Definition 6.27 (Sound unfolding).
Let $\beta = (N, \nu)$, $N = (B, E, G)$, be the unfolding of an acyclic system S. The *sound unfolding* of S is the maximal prefix of β that contains no event that is in causal relation either with a locally unsafe condition or a local deadlock in β. ⌟

Figure 6.33 exemplifies the sound unfolding of the net in Figure 6.31. The subnet of the unfolding below the dashed line must be pruned out because it contains

a lack of synchronization. Indeed, conditions c''_z, $c''_{z,e6}$, c''_{e6}, c'''_z, $c'''_{z,e6}$, and c'''_{e6} (highlighted with grey background) are locally unsafe. Accordingly, event $e_{\{B1,B2\}}$ (highlighted with black background) is the event that is causal with all locally unsafe conditions of the unfolding.

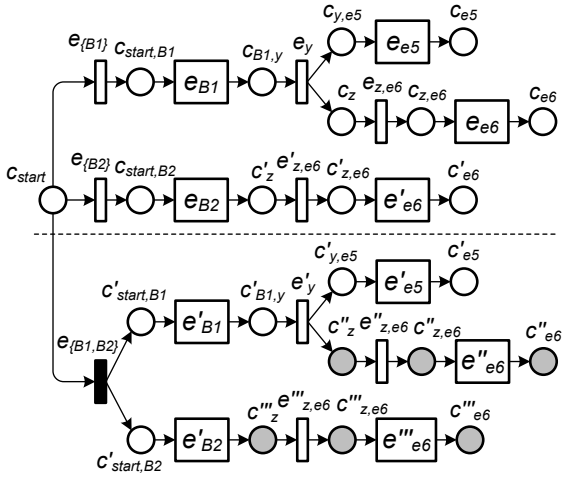


Figure 6.33. The sound unfolding of the net in Figure 6.31

In certain cases, some transitions of a system might be not represented in its sound unfolding. This happens when a transition does not occur in any execution that starts with a correct instantiation. Such systems are unsound and are excluded from further consideration.

Definition 6.28 (Soundness of acyclic nets).
Let $\beta = (N, \nu)$, $N = (B, E, G)$, be the sound unfolding of the augmented version $N' = (P, T, F)$ of an acyclic net N''. Let $S \subseteq T$ be the set of all start transitions of N'. N'' is said to be *sound*, iff for every $t \in T \setminus S$ there exists $e \in E$, such that $\nu(e) = t$. ⌟

Specifically, we say that a multi-source and multi-sink process model is sound if every non-start transition of the augmented version of the corresponding net appears in at least one execution that starts with a correct instantiation. The sound unfolding in Figure 6.33 represents all the non-start transitions of its originative net in Figure 6.31 and, hence, the model in Figure 6.29 is sound.

6.3.4. Structuring

Given a sound multi-source and multi-sink process model, one can construct an equivalent structured model using Algorithm 6 with the following changes:

- The corresponding net must be augmented according to Definition 6.25 (line 1).
- One must construct a proper prefix based on the sound unfolding, i.e., as a prefix of the sound unfolding (line 2).

Note that the sound unfolding of the net in Figure 6.31 and its proper prefix coincide, see above the dashed line in Figure 6.33.

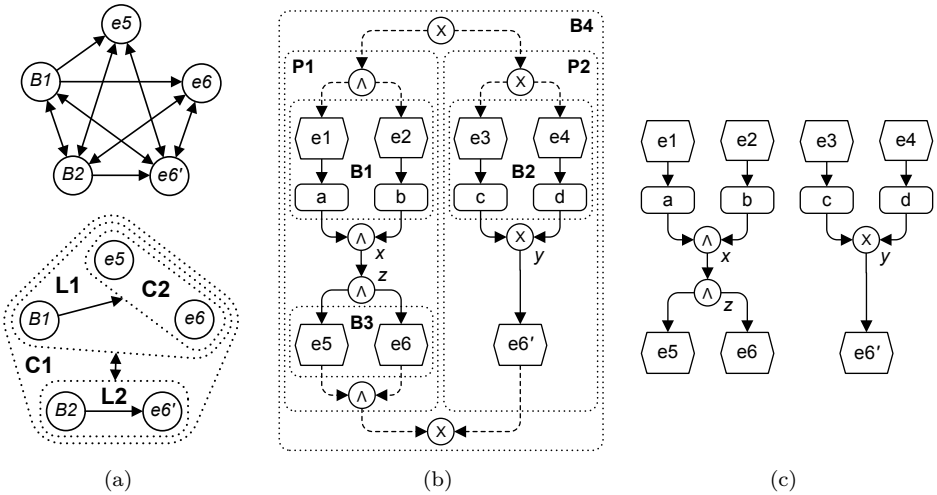


Figure 6.34. Structuring of the EPC in Figure 6.29(a): (a) the orgraph and its MDT, (b)–(c) structured versions of the EPC

Figure 6.34(a) shows the orgraph of the proper prefix in Figure 6.33, along with its MDT. The MDT contains no concurrent primitive and, therefore, the well-structured process model exhibiting given ordering relations exists. The resulting structured model has a single source and single sink, and starts and ends with gateways that encode instantiations and completions of the model, as shown in Figure 6.34(b), which is the well-structured version of the EPC in Figure 6.29(a). The figure also visualizes the process components: $B1$ and $B2$ are the components from Figure 6.29(b). Polygons $P1$ and $P2$ are synthesized from modules $L1$ and $L2$, respectively, see the MDT. Finally, bonds $B3$ and $B4$ correspond to modules $C1$ and $C2$, respectively. The gateways at the start and at the end can be trivially removed through a post-processing step, thereby yielding a structured multi-source and multi-sink EPC model, see Figure 6.34(c).

6.3.5. Evaluation

The proposed structuring method has been implemented in a tool named `bpstruct`, publicly available[5]. Using this implementation, we conducted an empirical evaluation of the proposed method with the aim of addressing the following questions in the context of a repository of process models taken from commercial practice:

Q1. What proportion of unstructured process models are inherently unstructured and what proportion of models are structurable?

[5] `https://code.google.com/p/bpstruct/`

Q2. Is the exponential worst-case complexity of the structuring method (particularly the unfolding method) problematic in practice?
Q3. In theory, the structuring method may lead to node duplication. To what extent does this duplication lead to larger process models?
Q4. The method for structuring multi-source models may lead to disconnected models. How often does this phenomenon occur?

In the following, we present the dataset which we used for the evaluation, and discuss answers to the questions proposed above, which we derived from the evaluation.

Dataset

For the evaluation, we used the SAP Reference Model [65] – a collection consisting of 604 EPCs capturing business processes supported by the SAP R/3 enterprise system. In the first stage, we discarded models that are already structured and models that contain cycles. We also discarded models that contained *or* joins in a rigid, since these models cannot be processed by the proposed method. Note that *or* joins at the boundaries of bonds do not pose any problem to `bpstruct`, since bonds are separated from rigids during the computation of the RPST, and `bpstruct` only needs to deal with rigids; a description of the execution semantics of *or* gateways can be found in [136]. After this pre-processing, we were left with 78 models, 40 of which are sound. As many models in the SAP Reference Model are multi-source and multi-sink, we checked soundness using the technique proposed in Section 6.3.3. Coincidently, each of the 40 sound models contained exactly one rigid component. Thus, the number of rigids that needed to be structured was also 40.

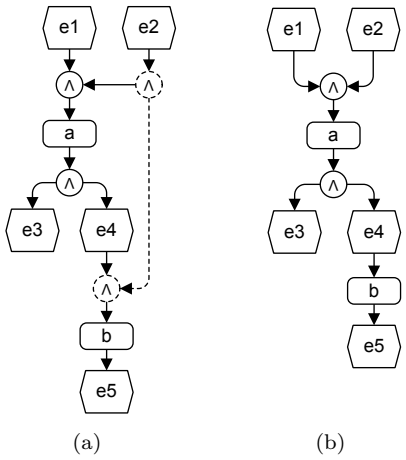


Figure 6.35. Structuring of a homogeneous *and* rigid

Among these 40 rigids, 6 are homogeneous *xor* rigids, 19 are homogeneous *and* rigids, and 15 are heterogeneous rigids. All homogeneous *xor* rigids can be structured, since the only source of inherent unstructuredness in acyclic process models stems from concurrency. On the other hand, all but one of the 19 homogeneous *and* rigids are inherently unstructured. The only *and* rigid that is structurable is a case of a rigid that contained redundant transitive arcs; the core structure of the rigid is summarized in Figure 6.35(a). The arc going from the *and* split after event *e2* to the *and* join after event *e4* is redundant and, thus, can be removed; in doing so, the split and the join become redundant and can also be removed. The structured version of the EPC in Figure 6.35(a) is proposed in Figure 6.35(b). All other 18 homogeneous *and* rigids contain the structure

depicted in Figure 5.6. Finally, among the 15 heterogeneous rigids, only 3 are inherently unstructured. The complete characterization of the dataset employed for the evaluation is depicted in Figure 6.36. From the total of 604 models, we did not address 31 EPCs with cycles and 96 EPCs with *or* gateways; these are depicted by empty circles in the figure. These cases are left for future work.

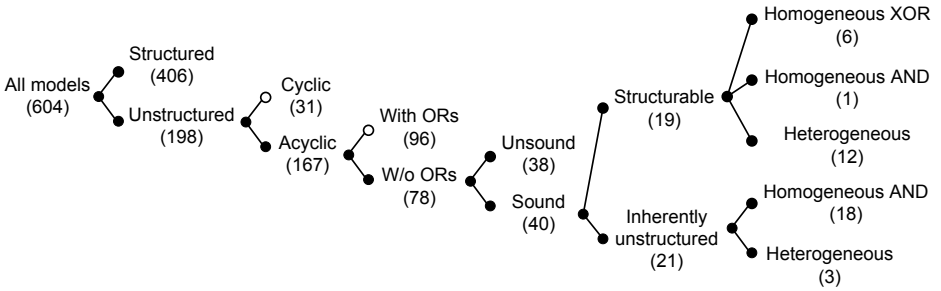
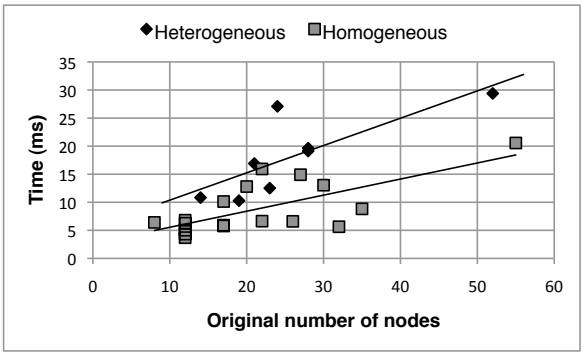


Figure 6.36. Structural characterization of models in the SAP Reference Model

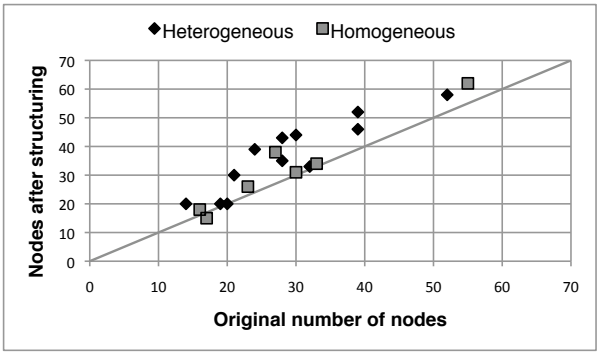
Results

Regarding question Q1 above, the evaluation suggests that unstructured homogeneous *and* rigids are highly prone to being inherently unstructured, while heterogeneous rigids are prone to be structurable. Regarding question Q2 above, Figure 6.37(a) plots the execution time relative to the size of the input model. The figure also plots two trendlines of the linear regression analysis: one for the heterogeneous and one for the homogeneous case. The plot shows that, although in theory the complexity of the structuring algorithm is exponential to the size of the input (due to the unfolding step), this worst-case exponential complexity does not manifest itself in the dataset at hand. The observed execution times are rather linear relative to the size of the input models. In the given dataset, we found one rigid component that contained 2 *xor* gateways intermingled with 8 *and* gateways. The structuring algorithm took 1.5 seconds to execute for this model and concluded that the model is inherently unstructured. These 1.5 seconds were spent mainly on the unfolding and the pruning steps. Putting this case aside, the average execution time for the remaining models is 12 ms (standard deviation: 7ms). These execution times exclude the time to compute the RPST, but this step has a linear complexity.

Regarding question Q3, Figure 6.37(b) plots the size of the structured process models relative to the size of the original models (only for models that were originally unstructured but not inherently unstructured). The figure shows that – except for the model with a homogeneous *and* rigid (located slightly below the diagonal), the size of the structured model is at least equal and in most cases larger than that of the input model. Specifically, we observed that the size of the output model (measured in terms of number of nodes) was up to 1.625 times that of the input models. On average, the size of the models produced by `bpstruct` was 1.22 times the size of the input model (standard deviation: 0.2). These results emphasize the fact that structuring a process model involves a tradeoff



(a) Average structuring time



(b) Duplication ratio

Figure 6.37. Experimental results

between modularity and size. Please note that duplication results were obtained for flat models, i.e., models without subprocesses. Structured models may also contain duplicates of whole process components, which leaves opportunities for modularization, e.g., by employing techniques like [125].

Finally, regarding question Q4, we found that 15 of the 19 structurable models led to disconnected structured models. Two of the rigids whose structured versions were connected originally had a single source, while two of them were multi-source. As expected, all rigids whose structured versions are disconnected were originally multi-source models. This result suggests that when structuring multi-source models, one is likely to obtain disconnected models. This phenomenon is specific to EPCs. It does not occur in the case of multi-source BPMN models, since in BPMN, the multiple disconnected fragments would be re-connected with a common event-driven or parallel event-driven gateway.

6.4. Towards Cyclic Structuring

A process model is *cyclic* if it contains at least one cyclic path, i.e., a path composed of more than one node that starts and ends at the same node, e.g.,

the path v, b, x, d, y, z, v in the process model in Figure 6.38. Every cyclic path of a process model has at least one entry node which can be either a *xor* join (node v, refer to Figure 6.38) or an *and* join (node x), and at least one exit node, that can be either a *xor* split (node y) or an *and* split (node z). Cycles without an entry cannot "start" and cycles without an exit cannot "terminate". A well-structured cyclic process model contains no rigid process components and, hence, every cyclic path is contained within some bond process component of the model. In a sound well-structured cyclic process model, it clearly holds that if a cyclic path contains the entry and the exit of bond process component $B1$, then the entry of $B1$ is a *xor* join and the exit of $B1$ is a *xor* split; we call $B1$ a *SESE loop* process component. Therefore, the task of transforming process models with arbitrary loops into process models where all cyclic paths are described by SESE loop components has to deal with transformations of cyclic paths embedded into rigid process components, e.g., the path v, b, x, d, y, z, v in rigid $R1$ in Figure 6.38, into paths that can be formalized with the help of bond process components.

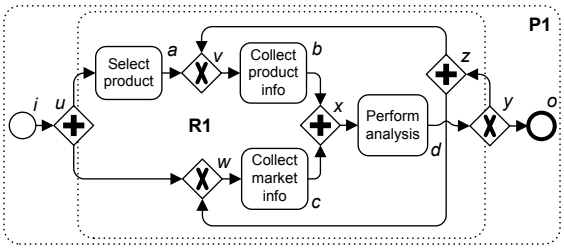


Figure 6.38. Unstructured cyclic process model

This section proposes a technique for structuring process models with cyclic paths. Similar to the previous sections, the technique is based on the notion of the proper complete prefix unfolding and rests on (i) the conjecture that proper prefixes are finite and (ii) sketches to the proofs along the individual steps of the technique; hence, the title of this section. In the following, we shall employ the process model in Figure 6.38 to exemplify some of the structuring steps.

6.4.1. Unfolding Cyclic Process Models

The main idea of our cyclic structuring technique is to unfold all cyclic paths of a process model to the point where structuring can be organized as a combination of the compiler techniques for structuring sequential programs and the approach for structuring acyclic process models proposed in Sections 6.1 and 6.2. The idea can be implemented if one possesses a technique for transforming an initial unstructured process model into an equivalent model in which all cyclic paths are captured either in SESE loop or non-concurrent rigid process components. To this end, we employ the notion of the proper complete prefix unfolding.

Importantly, in the case of process models with cyclic paths it must be shown that the algorithm for the construction of proper prefixes always terminates. The discussion proceeds based on the conjecture that the statement below holds.

Conjecture 6.1. *A proper complete prefix unfolding of a sound free-choice WF-system induced by an adequate total order for safe systems is finite.* ⋆

A proper complete prefix unfolding β_1 is an extension of the complete prefix unfolding β_2 of the WF-system. β_2 is finite. Every event in β_1 and not in β_2 is associated with a marking which is already associated with some event in β_2; note that β_2 is complete. The fact that one will eventually be able to construct an extension of β_2 truncated at healthy cutoff events can be deduced from the soundness of the WF-system (every concurrent run of a sound WF-system must be eventually synchronized).

Figure 6.39(a) shows the WF-net that corresponds to rigid component $R1$ in the process model in Figure 6.38, whereas Figure 6.39(c) shows its *finite* proper complete prefix unfolding. Unlike in the complete prefix unfolding, shown in Figure 6.39(b), in the proper prefix all the concurrency is kept encapsulated, refer to the discussion in Section 6.1.1. The complete prefix unfolding in Figure 6.39(b) provides just as much information which can be employed for structuring as its originative WF-net. In this particular example, the complete prefix unfolding can be seen as a kind of a spanning tree of the originative WF-net. If one *rewires* the complete prefix unfolding by merging conditions from $Cut(\lceil e_z \rceil)$ with the conditions from $Cut(\lceil e_a \rceil)$ based on the criterion that two conditions get merged if they refer to the same place of the originative net, one obtains the originative WF-net, e.g., in Figure 6.39(b) condition c'_v must be merged with condition c_v, whereas condition c'_w must be merged with condition c_w. However, the main problem of complete prefix unfoldings lies not even in the fact that sometimes prefixes do not exhibit additional structural information as compared to their originative systems; the main deficiency, as will be explained later, stems from the fact that they do not encapsulate concurrency within cyclic paths.

In the following, we refer to the net obtained by rewiring all the cutoff events of a complete prefix unfolding as the *rewired prefix*. It is easy to see that the order of different rewirings does not influence the final result.

The rewiring procedure, according to the principles discussed above for the complete prefix unfolding in Figure 6.39(b) (when one merges cuts of local configurations of cutoff events and corresponding events) can *sometimes* result in nets which are in a strong behavioral relation with their originative systems. More specifically, the rewired prefix and its originative system can be *occurrence net equivalent* [96, 157, 142], i.e., they can have isomorphic unfoldings, which simply means that both systems describe exactly the same behavior (even if these systems are structurally different). Trivially, the unfolding of the WF-net in Figure 6.39(a) and the unfolding of the net obtained by merging conditions $\{c'_v, c'_w\}$ and $\{c_v, c_w\}$ in the complete prefix unfolding in Figure 6.39(b), which is exactly the same as its originative net, are isomorphic. However, the WF-net in Figure 6.3(a) and the net obtained by merging conditions $\{c'_x, c'_y\}$ and $\{c_x, c_y\}$ in its complete prefix unfolding in Figure 6.3(b) do not have isomorphic unfoldings.

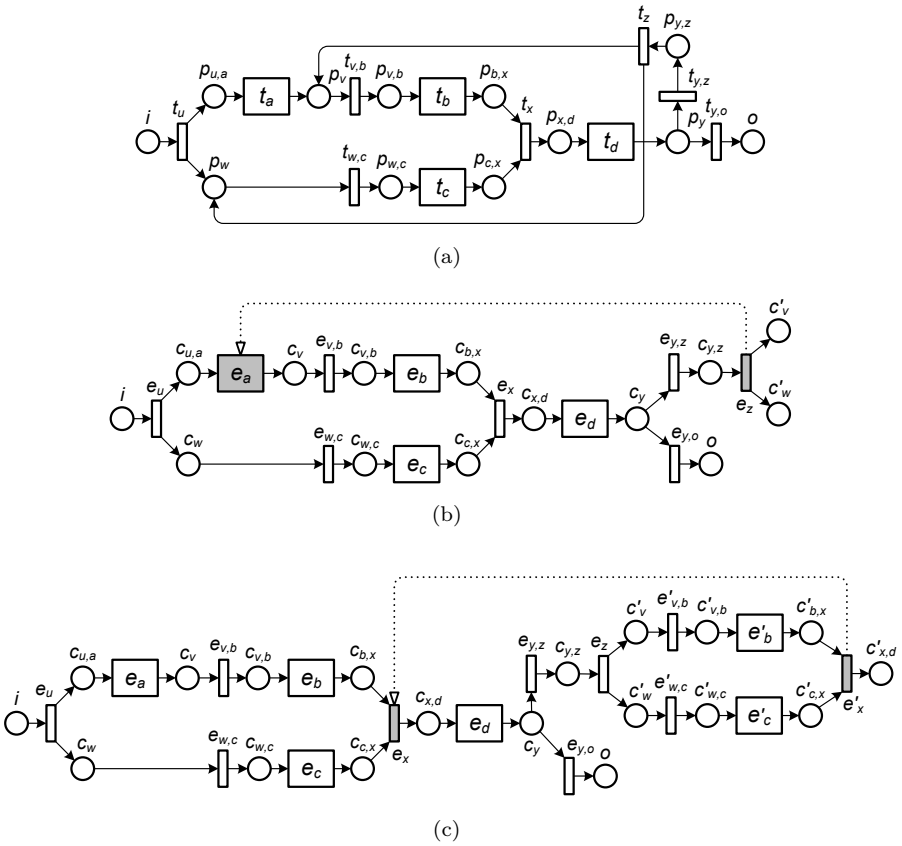


Figure 6.39. (a) WF-net that corresponds to rigid $R1$ of the process model in Figure 6.38, (b) the complete prefix unfolding of (a), and (c) the proper prefix of (a), both induced by an adequate total order for safe systems

Interestingly, if one rewires a proper complete prefix unfolding of a net system, the rewired prefix and the originative system are *always* occurrence net equivalent. In complete prefix unfoldings, a part of an unfolding which can be constructed after the cut induced by a local configuration of a cutoff event e is isomorphic with the part of the unfolding which follows after the cut induced by a local configuration of $corr(e)$ [38]. If e is healthy, the cuts of local configurations of e and $corr(e)$ differ only in post-conditions. Thus, the rewiring of a proper complete prefix unfolding can be accomplished by merging post-conditions of healthy cutoff events; this ensures that the unfolding of the resulting rewired net is isomorphic with the unfolding of the originative net.

Figure 6.40 shows the rewired prefix obtained from the proper complete prefix unfolding in Figure 6.39(c). Observe that the nets in Figure 6.39(a) and Figure 6.40

are structurally different, but have isomorphic unfoldings; please validate this claim by constructing unfoldings of both nets.

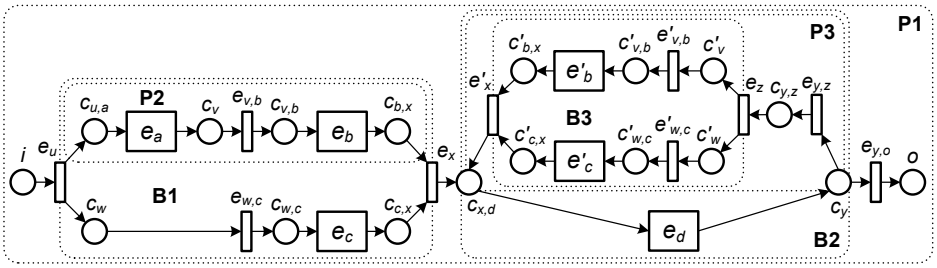


Figure 6.40. The rewired (proper) prefix constructed from the proper complete prefix unfolding in Figure 6.39(c)

For cyclic structuring purposes, we shall employ the rewiring of proper complete prefix unfoldings which gets a little bit more involved. In particular, we shall rely on the notion of the *rewired proper prefix*.

Definition 6.29 (Rewired proper prefix).
Let $\beta = (N, \nu)$, $N = (B, E, G)$, be a proper complete prefix unfolding of a net system S that corresponds to a process model and let $e \in E$ be a healthy cutoff event of β. The *rewiring* of e in β results in the *proper complete prefix unfolding* $\beta' = (N', \nu'), N' = (B', E', G')$*, of* S *with rewired cutoff event* e, where:

- If $|\, e\bullet\,| = 1$, then $B' = B \setminus e\bullet$, $E' = E$, $G' = (G \setminus (\{e\} \times e\bullet)) \cup (\{e\} \times corr(e)\bullet)$, and $\nu' = \nu|_{B' \cup E'}$.
- If $|\, e\bullet\,| > 1$, then $B' = B \setminus \{\bullet e \cup e\bullet\}$, $E' = E \setminus \{e\}$, $G' = (G \setminus ((\{e\} \times e\bullet) \cup (\bullet e \times \{e\}) \cup (\bullet(\bullet e) \times \bullet e))) \cup (\bullet(\bullet e) \times \bullet corr(e))$, and $\nu' = \nu|_{B' \cup E'}$.

β with all its healthy cutoff events rewired is called the *rewired proper prefix* of S.⌟

In rewired proper prefixes, the postsets of healthy cutoff events get merged with the postsets of corresponding events as follows: If a healthy cutoff event e has one post-condition, it gets merged with the only post-condition of $corr(e)$. However, if a healthy cutoff event e has more than one post-condition, we merge the only condition in $\bullet e$ with the only condition in $\bullet corr(e)$; event e and conditions in $e\bullet$ get removed. If compared with the initially introduced rewiring principles, in rewired proper prefixes healthy cutoff events with multiple post-conditions get special handling. Note that a healthy cutoff event e with multiple post-conditions corresponds to an *and* split in the process model and, hence, has a single pre-condition c. Furthermore, note that condition c is guaranteed to have at most one pre-event (unfoldings are captured by occurrence nets, see Definition 5.12) and one post-event (due to the mapping of *xor* splits to nets, see Definition 2.24). Therefore, given a proper complete prefix unfolding, one can always construct its rewired proper prefix.

It is easy to see that the rewired proper prefix of a WF-net is again a WF-net and that the originative WF-net and the rewired proper prefix are occurrence net equivalent. Observe that the WF-net in Figure 6.40 is the rewired proper prefix of the WF-net in Figure 6.39(a).

Cyclic paths in rewired proper prefixes have a nice property, i.e., the concurrency within these paths is kept encapsulated. If a cyclic path contains an event e with multiple post-conditions, then this cyclic path also contains an event e' with multiple pre-conditions such that every path that originates at e eventually reaches e' and every path that originates at a source place and visits e' also visits e. The property claimed above is supported by the following rationale: A proper complete prefix unfolding is acyclic – it is defined by an occurrence net, see Definition 5.12. Hence, cyclic paths can only be introduced when rewiring healthy cutoff events. Next, we take a closer look at all possible ordering relations of a healthy cutoff event e and its corresponding event:

- $corr(e) \# e$. If e and $corr(e)$ are in conflict, then there exists condition c which has two distinct events e' and e'' in its postset, such that (i) either $e' = e$ or $e' \rightsquigarrow e$ and (ii) either $e'' = corr(e)$ or $e'' \rightsquigarrow corr(e)$. Moreover, there exists no path from e' to $corr(e)$ and there exists no path from e'' to e; otherwise there exists a self-conflict in the proper prefix. Finally, if there exists event d with multiple post-conditions on a path from c to e, then every path that originates at d eventually reaches e, and if there exists event f with multiple pre-conditions on a path from c to e, then every path that originates at a condition without pre-events and visits f also visits c; otherwise one can easily show that e is not healthy. The same rationale applies to all the events with mutiple pre-/post-events on paths from c to $corr(e)$, i.e., concurrency is kept encapsulated between c and $corr(e)$.
- $corr(e) \rightsquigarrow e$. The ideas discussed for the situation when e and $corr(e)$ are in conflict applies with minor adjustments to the case when $corr(e)$ and e are in causal relation, i.e., the regions of concurrency on the paths from $corr(e)$ to e are kept encapsulated; otherwise e is not healthy.
- $corr(e) \parallel e$. If e and $corr(e)$ are concurrent, then there exist at least two distinct conditions in the postsets of e and $corr(e)$ which are concurrent and refer to the same place in the originative system. This contradicts with the assumption that the originative system is safe, see Proposition 6.3 in [38]. Therefore, the situation when $corr(e) \parallel e$ is not possible.

By stepwise applying individual rewirings of healthy cutoff events in proper prefixes we keep concurrency encapsulated which allows for later modularization of the structuring problem, which will be exploited in the next section. It must be noted that the construction principles for the rewired proper prefixes are specifically designed to support later modularization of the structuring problem. Intuitively, we introduce special handling when rewiring healthy cutoff events with multiple pre-conditions in order to minimize the amount of events which can introduce concurrency on cyclic paths in the resulting nets.

6.4.2. Cyclic Structuring Algorithm

The rewired proper prefix in Figure 6.40 already corresponds to the well-structured process model in Figure 6.41(a); the process model can be trivially constructed by following the principles proposed in Section 6.2.8.

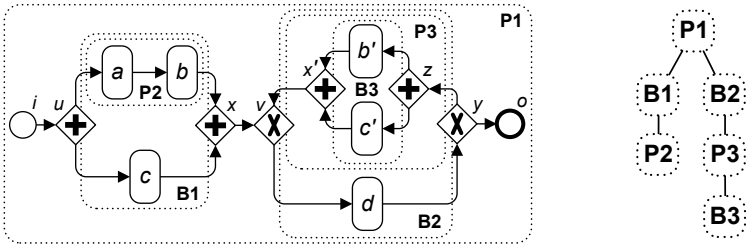


Figure 6.41. (a) The well-structured version of the process model in Figure 6.38, and (b) the (simplified) RPST of (a)

However, in general, the rewired proper prefix by itself is not sufficient to accomplish structuring of a cyclic process model. It is more precisely to say that the rewired proper prefix delivers opportunities for modularization of the structuring problem. The modularization can be organized by using the RPST of the rewired proper prefix and abstraction of cyclic paths. Algorithm 7 summarizes the technique for structuring sound cyclic process models. Next, we exemplify and discuss every step of the algorithm in details. To support the discussion, we shall use the process model in Figure 6.42 as our running example.

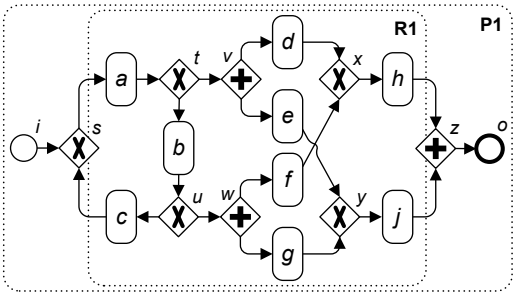


Figure 6.42. Unstructured cyclic process model

Algorithm 7 starts by constructing the rewired proper prefix of a given sound cyclic process model (lines 1–3); the constructions can be accomplished by employing Definition 2.24, Definition 6.1, and Definition 6.29. Figure 6.43 shows the rewired proper prefix of the WF-net that corresponds to rigid $R1$ from the process model in Figure 6.42. Note that WF-net N (constructed at line 1 of the

Algorithm 7: Structuring cyclic process model

Input: A sound cyclic process model PM
Output: An equivalent well-structured process model

1 Construct WF-net N that corresponds to PM
2 Construct proper complete prefix unfolding β of N induced by an adequate total order for safe systems
3 Construct γ – the rewired proper prefix of N – from β
4 Compute SCC – the set of all strongly connected components of γ
5 Compute $SEME$ – the set of SEME-cyclic subnets from components in SCC
6 Construct γ' by abstracting subnets from $SEME$ in γ
7 Construct well-structured process model PM' from γ' by using Algorithm 6
8 Structure maximal acyclic subnets in $SEME$ using Algorithm 6
9 Construct PM'' by refining PM' with well-structured subnets from $SEME$
10 Structure *xor* rigids in PM'' by using compiler techniques, e.g., [97]
11 **return** PM''

algorithm) and rewired proper prefix γ (constructed at line 3) define the same behavior, i.e., unfoldings of both nets are isomorphic.

The algorithm proceeds at line 4 by computing the set of all *strongly connected components* of γ. The strongly connected components of a directed graph are its maximal strongly connected subgraphs. A directed (sub)graph is called strongly connected if there is a directed path from every vertex in the graph to every other vertex. The net in Figure 6.43 contains one strongly connected component. This component is defined by a subnet induced by the nodes highlighted with grey background in Figure 6.43.

Every strongly connected component of the rewired proper prefix captures a subset of its cyclic paths; recall from the previous section that concurrency within these paths is kept encapsulated. Therefore, every strongly connected component can be seen as an area of sequential execution within the rewired proper prefix,

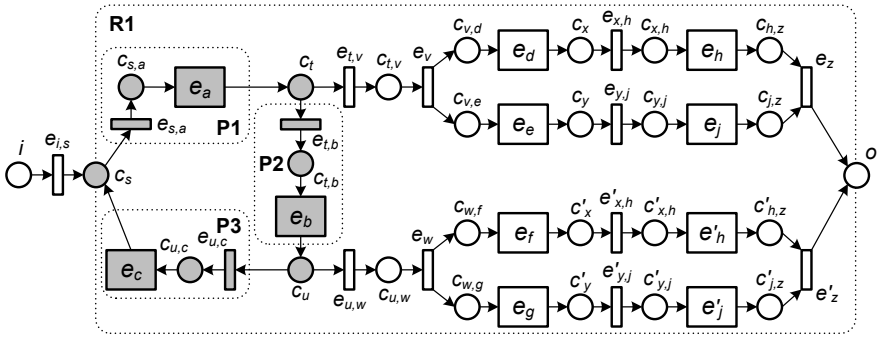


Figure 6.43. The rewired proper prefix of the WF-net that corresponds to rigid process component $R1$ from the process model in Figure 6.42

which gives an opportunity for its abstraction. Indeed, the number of tokens that can simultaneously reside at the *core* places of a strongly connected component, i.e., places c_s, c_t, and c_u of the only strongly connected component of the net in Figure 6.43, is never larger than one; otherwise the net is unsafe. The *core* places of a strongly connected component are the places shared by the strongly connected component and the triconnected component of the rewired proper prefix that is used to derive the smallest (in the number of arcs) canonical fragment which contains the strongly connected component; see Section 3.3 for details on correspondences of the triconnected components and canonical fragments of a TTG. In Figure 6.43, the smallest canonical fragment which contains the only strongly connected component is fragment $R1$, which corresponds to the triconnected component of the net composed of nodes c_s, c_t, c_u, and o. Note that core places of a strongly connected component are not necessarily the boundary places of the strongly connected component.

Algorithm 7 proceeds by abstracting strongly connected components. Every strongly connected component gets abstracted within its single-entry-multi-exit-cyclic (SEME-cyclic) subnet. The only strongly connected component in the net in Figure 6.43 has one entry (place c_s) and two exits (places c_t and c_u); the nodes can be classified using Definition 3.2. Hence, the strongly connected component defines a SEME-cyclic subnet of the rewired proper prefix. However, in general, a strongly connected component of the rewired proper prefix can have multiple entries and/or multiple exits. If a strongly connected component has multiple entries, we extend the component to a single entry subnet prior to proceeding with its abstraction. A strongly connected component can be extended to a SEME-cyclic subnet by including nodes and arcs up to the common dominator of all its entry nodes. Technically, this can be accomplished by employing techniques from the compiler theory, e.g., [126].

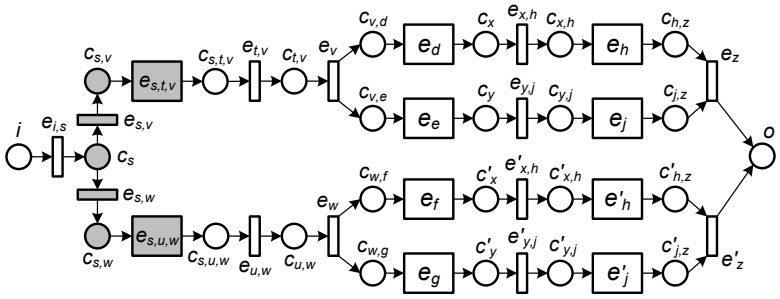


Figure 6.44. The rewired proper prefix from Figure 6.43 with abstracted SEME-cyclic subnets

Once all the SEME-cyclic subnets that correspond to the strongly connected components of the rewired proper prefix are identified at line 5 of Algorithm 7, they get abstracted within the rewired proper prefix at line 6. The abstraction proceeds as follows: A SEME-cyclic subnet gets contracted into a single place.

Fresh observable *decision transitions* are inserted in the net following each of the outgoing arcs of the contracted place. Figure 6.44 shows the rewired proper prefix from Figure 6.43 with its only SEME-cyclic subnet abstracted. In the figure, the SEME-cyclic subnet from Figure 6.43 is contracted into place c_s. The token at the contracted place hints at the fact that the SEME-cyclic subnet is executing. Transitions $e_{s,t,v}$ and $e_{s,u,w}$ are the fresh decision transitions. They are introduced in order to keep references to the exits of the abstracted SEME-cyclic subnet.

A WF-net obtained by abstracting all the SEME-cyclic subnets in a rewired proper prefix is acyclic and, hence, its structuring can be accomplished by the techniques proposed in Section 6.1 and Section 6.2. This is done at line 7 of Algorithm 7. Note that one should provide a WF-net, rather than a process model, as input to the structuring algorithm. During structuring, decision transitions must be treated as observable transitions. Figure 6.45 shows the process model constructed from the WF-net in Figure 6.44 by providing the net as input to Algorithm 6 (starting from line 2).

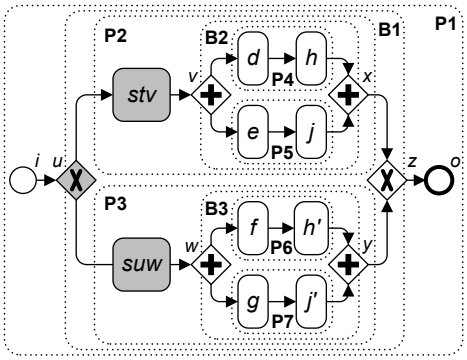


Figure 6.45. Well-structured process model constructed from the WF-net in Figure 6.44

The process model in Figure 6.45 is well-structured. However, it contains no cyclic paths of the original unstructured process model (see in Figure 6.42). The next steps of Algorithm 7 deal with refining the well-structured acyclic process model obtained at line 7 of Algorithm 7 with cyclic paths of the input process model. Well-structured process models constructed by our structuring techniques from Section 6.1 and Section 6.2 do not keep references to the conditions of proper prefixes that they are constructed from. For instance, the process model in Figure 6.45 keeps no reference to the contracted place c_s in Figure 6.44. However, the process model in Figure 6.45 keeps references to the decision transitions of the net in Figure 6.44, see tasks *stv* and *suw*. These tasks allow for the unique identification of points in the process model which must be refined with the priorly abstracted SEME-cyclic subnets. Before proceeding with refinements, every SEME-cyclic subnet must be structured. Structuring of a SEME-cyclic subnet can be organized by structuring canonical fragments of the rewired proper prefix which are contained in the subnet. Note that the structure of the SEME-cyclic subnet given by its core places stays intact. For instance, structuring of the only SEME-cyclic

subnet in Figure 6.43 can be accomplished by structuring canonical fragments $P1$, $P2$, and $P3$, which collectively compose the subnet. SEME-cyclic subnets are structured at line 8 of Algorithm 7. Figure 6.46 shows the result of line 9 of Algorithm 7; the process model is obtained by refining the model in Figure 6.45 with the well-structured version of the only SEME-cyclic subnet from Figure 6.43.

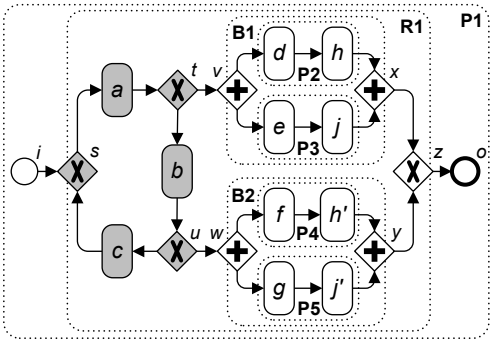


Figure 6.46. Process model constructed from the process model in Figure 6.42 by applying Algorithm 7 up to and including line 9

For the process models constructed at line 9 of Algorithm 7, it holds that all cyclic paths are captured within *xor* rigids. *xor* rigids can be structured by employing standard compiler techniques, e.g., [97]. For example, the process model in Figure 6.46 contains one *xor* rigid $R1$ which still must be transformed in order to finalize the structuring of the input process model. Structuring of *xor* rigids is accomplished at line 10 of Algorithm 7. Finally, the resulting well-structured process model is returned at line 11 of the algorithm.

In this section, we described the algorithm for structuring cyclic process models. The algorithm uses the RPST of the rewired proper prefix and abstraction of cyclic paths in order to modularize the problem of structuring cyclic process models into many smaller problems of structuring acyclic process models and sequential programs. In the example which we used to explain the algorithm, the WF-net constructed at line 6 of the algorithm, shown in Figure 6.44, and the WF-nets which are subjects for structuring at line 8 (see fragments $P1$, $P2$, and $P3$, in Figure 6.43), are rather trivial. However, in general, WF-nets processed within Algorithm 7 can be arbitrarily complex; which means they can as well be inherently unstructured. If one encounters such a WF-net, one can either decide to apply the technique for maximal-structuring from Section 6.2 or to report that the input model is inherently unstructured.

6.5. Conclusion

In this chapter, we have proposed several techniques for structuring process models. Each technique is designed to address structuring of a specific class of unstructured process models. Yet, all the techniques are founded on the basic

technique for acyclic structuring, see Section 6.1. We conclude that a sound acyclic process model is inherently unstructured if and only if its RPST contains a rigid process component for which the modular decomposition of its orgraph contains a concurrent primitive. In all other cases, Algorithm 6 applied to every rigid process component of a process model constructs its FCB-equivalent well-structured process model. We have thus provided a characterization of the class of well-structured acyclic process models under fully concurrent bisimulation, and a complete structuring method.

The method of acyclic structuring stops when the input model contains an inherently unstructured process component. Section 6.2 complements the technique of acyclic structuring by providing a method to synthesize the components corresponding to inherently unstructured parts of the input process model. This allows one to accomplish the partial structuring of an acyclic process model or to construct its FCB-equivalent maximally-structured process model.

In Section 6.3, we removed the restriction on the number of source and/or sink nodes in a process model by proposing a method for structuring acyclic multi-source and multi-sink models. This required us to generalize the notions of structuredness and soundness, as well as to define a precise instantiation semantics, for process models with multiple source nodes.

All the techniques mentioned above can also be used to structure process models with SESE cycles, even if these cycles contain unstructured components. In this case, the unstructured components and the SESE cycles are handled separately within the RPST of the input process model. However, these techniques cannot deal with models that comprise arbitrary cycles. In Section 6.4, we discussed the technique for structuring process models with cyclic paths. The technique summarizes our best practice for structuring cyclic process models. We believe that future developments of this technique can lead to a solid approach for structuring process models with arbitrary cycles.

The notion of behavioral equivalence that we employ for structuring, i.e., fully concurrent bisimulation, cares only about the opportunity to extend equivalent runs of process models (corresponding systems) with the same task (observable transition) possibly by skipping several gateways (places and/or silent transitions), see Definition 5.11. Technically, this implies that our structuring techniques might suppress implicit decision points (a *xor* split followed by another gateway) of an unstructured process model in its FCB-equivalent well-structured version. To avoid aggregation of several implicit decisions into a single decision, one should materialize them, i.e., introduce observable *decision tasks*, each following every outgoing arc of a *xor* split. Decision tasks are designed to represent decisions explicitly and should be treated as all other observable tasks during structuring. Afterwards, decision tasks can be converted back into the (implicit) decision points in the resulting structured process model.

One of the motivating reasons for structuring process models is to be able to apply existing analysis techniques which only work for structured process models. For example, existing techniques for calculating Quality of Service (QoS) properties of business processes [75] are only applicable for structured process models. In a separate work [26], we have shown how these techniques can be extended to models

with arbitrary topology by applying structuring in conjunction with additional post-processing steps to deal with inherently unstructured rigid components.

Another potential benefit of structuring a process model is that the resulting structured model may be easier to comprehend and less error-prone to maintain thanks to its higher degree of modularity [76]. However, the empirical evaluation reported in this article has put into evidence that the structured process models are often larger than the original ones (by an average of 22% in the case of the SAP Reference Model). Larger models are generally more difficult to comprehend and maintain than smaller models. An interesting direction for future work is to conduct empirical evaluations with end-users in order to determine whether the complexity reduction due to the modularity of the structured model outweighs the increase in complexity due to the larger size of the structured model.

All the structuring techniques proposed in this thesis are implemented in the `bpstruct` tool, which is publicly available. The running time of the structuring techniques is mostly dominated by the time required to compute proper prefixes, which for safe systems has an upper bound of $O(|T| \cdot R^{\xi})$, where T is the set of transitions, R is the number of reachable markings, and ξ is the maximal size of the presets or postsets of the transitions in the originative system [38]. All other steps can be accomplished in low polynomial or linear time. Concerning the extension for maximal structuring, the theoretic discussion in this thesis implies exponential time and space complexities when constructing posets (this is due to our intent to stay close to the existing theory). However, in practice, given an ordering relations graph one can construct a poset which only contains information from the orgraph, without introducing duplicate events, and thus stay linear to the size of the orgraph. At the theoretical level this requires the introduction of a concept of a cutoff for posets followed by an adjustment of the theories along subsequent transformation steps, see Section 6.2. The folding step is a reverse of the unfolding and, thus, in the best case can be performed in the same time.

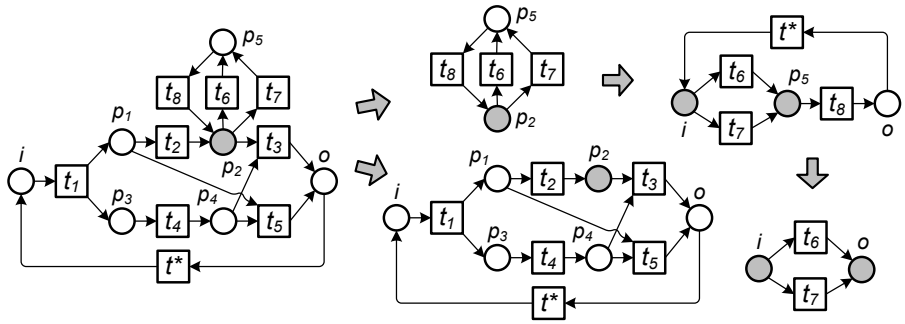
The fact that the running time of structuring depends on the size of the result allows the introduction of a heuristic to terminate computation if the result gets large, e.g., the event duplication factor in a proper prefix gets larger than two. Moreover, we envision a technique which can decide on-line, i.e., during the construction of the proper prefix, that from now on the prefix defines an ordering relations graph which contains a primitive module and, thus, the model cannot be structured. However, in practice we have never observed such a need with our implementation in most cases delivering results in milliseconds. Note that the amount of task duplication in the structured models is controlled by proper prefixes. As proper prefixes are always minimal, see the discussion in Section 6.1.1, the structuring introduces the minimal duplication whenever required.

Close to our maximal structuring setting, the problem of synthesizing nets from behavioral specifications has been a line of active research for about two decades [7, 18]. This area has given rise to a rich body of knowledge and to a number of tools, in particular `viptool` [7] and `petrify` [18]. Yet, these solutions fail in our setting: `petrify` aims at maximizing concurrency while our synthesis preserves given concurrency, `viptool` synthesizes nets with arc weights, which do not map to process models.

Part IV.

Analysis

7. Stepwise Connectivity-Based Verification of WF-nets



In this chapter, we study the relation between the connectivity property of short-circuit nets and the correctness, i.e., soundness, of WF-nets. Section 7.1 introduces principles of structural verification employed in the chapter. WF-nets are in tight relation with the class of strongly connected Petri nets. We emphasize this relation in Section 7.2. Afterwards, Section 7.3 presents basic connectivity-related properties of nets. Then, in Sections 7.4 to 7.6, we investigate how the connectivity of a short-circuit net can be used for reasoning about the soundness of the corresponding WF-net. We perform the stepwise connectivity-based decomposition of a short-circuit net and derive conclusions about the soundness of the WF-net. At the same time, we discuss the computational complexity of algorithms that support the conclusions. For each decomposition step we discuss techniques for performing decompositions, methods for performing soundness, and feedback that can be provided to process analysts in cases of unsoundness. Section 7.7 reports on the practical experience gained from the application of the theoretical results. Finally, Section 7.8 discusses related work and draws conclusion.

The materials reported in this chapter are published in [114, 115].

7.1. About Structural Verification

Behavioral models capture operational principles of real-world or designed systems. Formally, each behavioral model defines the state space of a system, i.e., its states and the principles of state transitions. Such a model is the basis for analysis of the system's properties. In practice, state spaces of systems are immense, which results in huge computational complexity for their analysis. Behavioral models are typically described as executable graphs, whose execution semantics encodes a state space. The structure theory of behavioral models studies the relations between the structure of a model and the properties of its state space.

In this chapter, we use the connectivity property of graphs to achieve an efficient and extensive discovery of the compositional structure of behavioral models; behavioral models get stepwise decomposed into components with clear structural characteristics and inter-component relations. At each decomposition step, the discovered compositional structure of a model is used for reasoning on properties of the whole state space of the system. The approach is exemplified by means of a concrete behavioral model and verification criterion. That is, we analyze workflow nets (Definition 2.20), a well-established tool for modeling behavior of distributed systems, with respect to the soundness property (Definition 2.21), a basic correctness property of workflow nets.

Soundness verification is a well-studied problem. A basic technique for soundness verification is state space exploration. However, state space exploration suffers from the state space explosion problem, as the number of reachable states can be exponential in the size of the model. In the following, we perform structural analysis of workflow nets to investigate soundness, providing insight into an alternative – and in many cases, preferable – way to check soundness. A workflow net gets decomposed into components based on its separating sets, i.e., sets of nodes of the net that when removed yield the net disconnected. We proceed stepwise, i.e., by gradually increasing the size of investigated separating sets. Based thereon, we point out how the soundness verification can be organized from the derived components of a workflow net. Where applicable, we draw conclusions on soundness for the general class of workflow nets; otherwise, the results are restricted to safe or free-choice nets. At each step of the decomposition, we argue about the algorithmic complexity of reaching the step and suggest diagnostic information that can be presented to process analysts as feedback on flaws in the behavior of a model.

Stepwise verification allows the detection of violations of the soundness property by inspecting small portions of a model, thereby considerably reducing the amount of work to be done to perform soundness checks. Though recent works show impressive results in efficiency of the soundness verification of industrial models, cf., [41, 42], the results were achieved under the assumption of free-choiceness of models. Efficient soundness verification in the case of general nets will, however, allow one to verify models which contain advanced workflow patterns. Despite the variety of existing soundness verification techniques, the efficiency and the structural diagnostic information for the general class of workflow nets are unique characteristics of our approach, coming at the expense of verification completeness.

We see a great potential in combining our approach with existing techniques on soundness verification to achieve more mature and complete verification.

7.2. Strong Connectivity of WF-nets

The structure of a Petri net (P, T, F) is defined by the graph $(P \cup T, F)$. There is an interesting observation that relates the behavioral characteristics of a net with the connectivity of the underlying graph. The *strong connectedness theorem* [22] states that a net N for which there exists a marking M_0, such that (N, M_0) is live and bounded, is strongly connected. In other words, strong connectedness of a short-circuit net is a mandatory condition for soundness of the corresponding WF-net, as the latter is traced back to liveness and boundedness of the respective short-circuit net. A net is *strongly connected*, if there exist a directed path from each node in the net to each other node. Though implied by soundness, the requirement of strong connectedness of short-circuit nets is explicitly stated in the definition of WF-nets. Note that strong connectedness of a net can be tested in the time linear to the size of the net, viz. $O(|P \cup T| + |F|)$, by using Tarjan's algorithm for discovery of strongly connected components of a graph, cf., [123].

Next, we examine connectivity-related properties of short-circuit nets, those not required by Definition 2.20, which can be used to explain soundness of WF-nets.

7.3. Connectivity of WF-nets

In this section, we briefly recall connectivity-related properties of graphs, see Section 3.2.1. We narrow down the discussion to those properties which we shall check later on short-circuit nets to derive conclusions about the soundness of the WF-nets. Detailed discussions and extensive examples will be provided as we proceed with the presentation of main results.

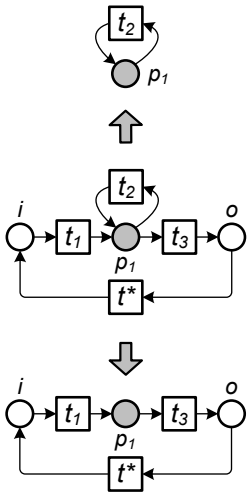


Figure 7.1. Decomposition of a short-circuit net

Two nodes x and y of a net are *connected*, if there is a path between x and y. Note that here we refer to a path that ignores directions of flow arcs. A net is *connected*, if every pair of distinct nodes of the net is connected. *k-connectivity* is the generalization of the connectivity property of a net. A net is *k-connected* if there exists no set of $k-1$ nodes, whose removal renders the net *disconnected*, i.e., there is no path between some pair of nodes in the net. The set of nodes whose removal disconnects the net is called a *separating* $(k-1)$-*set* of the net. Separating 1- and 2-sets are called *cutvertices* and *separation pairs*, while 1-, 2-, and 3-connected nets are referred to as *connected*, *biconnected*, and *triconnected*, respectively. The *connectivity* of a net is the largest k for which the net is k-connected.

Figure 7.1 shows a short-circuit net (in the middle). Short-circuit nets are strongly connected and, thus, are connected. However, a short-circuit net must not be k-connected, where k is larger than one. For instance, the net in Figure 7.1 contains cutvertex p_1 (highlighted with grey background). Thus, the net is *not* biconnected. Separation sets of a net can be used to decompose the net into subnets, or *components*, of a higher connectivity. The removal of the only cutvertex in the net in Figure 7.1 causes two subnets, one induced by nodes $\{p_1, t_2\}$ and the other induced by nodes $\{i, t_1, p_1, t_3, o, t^\star\}$ (at the top and at the bottom of the figure, respectively). Both subnets contain no cutvertices and, hence, are biconnected. In the subsequent sections, we study how separating sets and subnets caused by these sets can be used to decide on soundness of WF-nets.

7.4. The Biconnected Step

As already explained in the previous section, short-circuit nets are connected, but not necessarily biconnected. This section shows how the soundness verification of a WF-net can be broken down into checks of biconnected components of its short-circuit net.

7.4.1. Biconnected Decomposition of a WF-net

The classic sequential algorithm for computing biconnected components in a connected graph (Section 3.2.2), proposed in [53], runs in linear time. Let (V, E) be a connected graph, then the algorithm requires time and space proportional to $max(|V|, |E|)$. Biconnected components can be arranged in a tree structure – the *tree of the biconnected components*. The tree has two types of nodes that refer either to cutvertices or to biconnected components. Edges of the tree connect cutvertices with associated biconnected components, i.e., there is an edge between a cutvertex and a biconnected component, if and only if the biconnected component contains the cutvertex. The number of nodes in the tree is $O(|V|)$ and, hence, the space required to store the tree and all the biconnected components is $O(max(|V|, |E|))$, i.e., linear to the size of the original graph. Such a tree is also known as BC-tree, cf., [5].

The results obtained for graphs can be transferred to WF-nets. A *biconnected subnet* of a WF-net is a biconnected component of its short-circuit net. The *tree of the biconnected subnets*, or the *2-WF-tree*, of a WF-net is the tree of the biconnected components of its short-circuit net.

Definition 7.1 (The tree of the biconnected subnets).
Let $N = (P, T, F)$ be a WF-net and let N' be its short-circuit net with the extra transition t^*. The *tree of the biconnected subnets*, or the *2-WF-tree*, of N is a tuple $\mathcal{T}_N^2 = (\mathcal{B}, \mathcal{C}, \rho, \eta, \triangle)$, where:

- $\mathcal{B}$ is the set of all biconnected subnets of N',
- $\mathcal{C}$ is the set of all cutvertices of N',
- $\rho = (P_\rho, T_\rho, F_\rho) \in \mathcal{B}$ is the *root* of $\mathcal{T}_N^2$, iff $t^* \in T_\rho$,

- $\eta : \mathcal{B} \to \mathcal{P}(\mathcal{B})$ assigns to each biconnected subnet its child biconnected subnets; for subnet $b_1 \in \mathcal{B}$, b_1 is a *parent* of $b_2 \in \mathcal{B}$ and b_2 is a *child* of b_1, iff $b_2 \in \eta(b_1)$, and
- $\triangle \subseteq \mathcal{B} \times \mathcal{C} \times \mathcal{B}$, $(b_1, c, b_2) \in \triangle$, iff c is shared by b_1 and b_2, and $b_2 \in \eta(b_1)$. ⌟

Observe that Definition 7.1 captures the result of applying some algorithm for computing biconnected components on a short-circuit net. Note also that we deliberately choose biconnected subnet that contains the extra transition t^* as the root of the 2-WF-tree; there is always one such subnet. Function η must be defined iteratively starting from the root subnet, i.e., subnet ρ has no parent, each child subnet of the root subnet shares a cutvertex with it, each child subnet of the root subnet is the parent of subnets that it shares a cutvertex with (except its parent), etc.

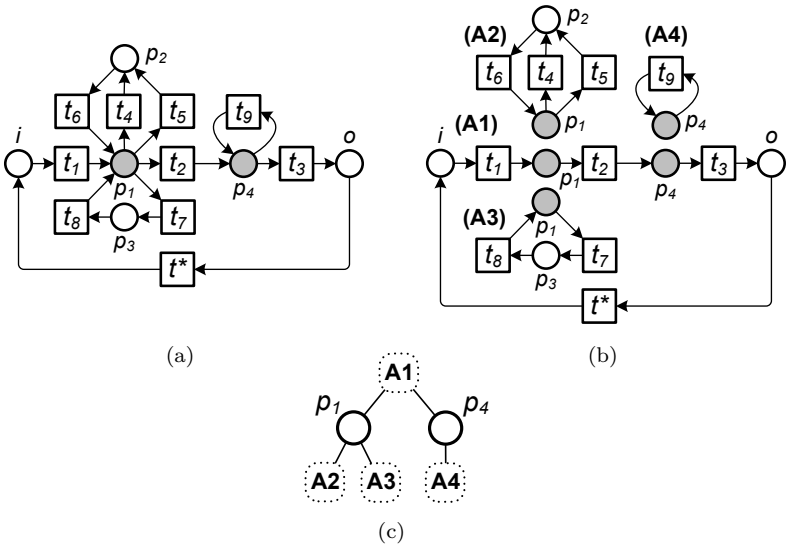


Figure 7.2. (a) A short-circuit net, (b) biconnected subnets, and (c) the 2-WF-tree

Figure 7.2 exemplifies the biconnected decomposition of a WF-net: Figure 7.2(a) shows a short-circuit net. The net has two place cutvertices: $p1$ and $p4$; both are highlighted with grey background. The cutvertices induce four biconnected subnets $A1$–$A4$, see Figure 7.2(b). Finally, Figure 7.2(c) organizes the subnets in the 2-WF-tree with the root node that corresponds to the biconnected subnet $A1$.

Each biconnected subnet of a WF-net can be seen as a self-contained part of the whole net which itself can be formalized as a WF-net, referred to as a biconnected sub-WF-net of the original WF-net.

Definition 7.2 (Biconnected sub-WF-net).
Let $N = (P, T, F)$ be a WF-net, $\mathcal{T}_N^2 = (\mathcal{B}, \mathcal{C}, \rho, \eta, \triangle)$ its 2-WF-tree, and $b = (P_b, T_b, F_b) \in \mathcal{B}$ its biconnected subnet. A *biconnected sub-WF-net* of N, denoted by $b^\star$, $b^\star = (P_{b^\star}, T_{b^\star}, F_{b^\star})$, is a net, such that:

- If $b = \rho$, then $P_{b^\star} = P_b$, $T_{b^\star} = T_b \cap T$, and $F_{b^\star} = F_b \cap F$.
- If $b \neq \rho$ and $a \in \mathcal{B}$, $c \in \mathcal{C}$ are such that there exists $(a, c, b) \in \triangle$, then
  - if $c \in P$, then $P_{b^\star} = (P_b \setminus \{c\}) \cup \{i, o\}$, $T_{b^\star} = T_b$, and $F_{b^\star} = \{(x_1, x_2) \in F_b \mid x_1 \neq c \wedge x_2 \neq c\} \cup \{(i, x) \in \{i\} \times T_b \mid (c, x) \in F_b\} \cup \{(x, o) \in T_b \times \{o\} \mid (x, c) \in F_b\}$.
  - if $c \in T$, then $P_{b^\star} = P_b \cup \{i, o\}$, $T_{b^\star} = (T_b \setminus \{c\}) \cup \{t_i, t_o\}$, and $F_{b^\star} = \{(x_1, x_2) \in F_b \mid x_1 \neq c \wedge x_2 \neq c\} \cup \{(i, t_i), (t_o, o)\} \cup \{(t_i, x) \in \{t_i\} \times P_b \mid (c, x) \in F_b\} \cup \{(x, t_o) \in P_b \times \{t_o\} \mid (x, c) \in F_b\}$.

A WF-net that corresponds to a subtree of b, denoted by $b^\triangle$, can be obtained by merging (at shared cutvertices) sub-WF-net $b^\star$ with all biconnected subnets that are descendants of b in the 2-WF-tree. Biconnected sub-WF-nets are also referred to as *biconnected WF-nets*.

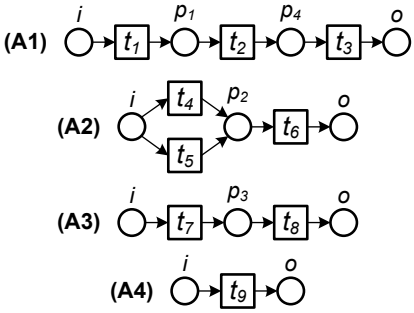


Figure 7.3. Biconnected sub-WF-nets

Figure 7.3 presents biconnected sub-WF-nets of the short-circuit net that is given in Figure 7.2(a). The sub-WF-nets correspond to the biconnected subnets in Figure 7.2(b). Sub-WF-net $A1$ is obtained from the corresponding biconnected subnet by ignoring transition t^* and arcs that are incident to t^*. In the case when a biconnected subnet is not the root of the 2-WF-tree, see Figure 7.2(c), the corresponding sub-WF-net can be obtained as follows: The cutvertex that corresponds to the parent node of the biconnected subnet in the 2-WF-tree is removed from the subnet and two fresh places are added; these are source place i and sink place o. If the removed cutvertex was a place, then all its outgoing arcs get rerouted to originate from i, whereas all its incoming arcs are rerouted to terminate at o. If the removed cutvertex was a transition, then two additional fresh transitions are added; these are transitions t_i and t_o. Afterwards, the flow relation is extended, such that t_i is put in the postset of i, while t_o is put in the preset of o. Finally, outgoing arcs of the removed cutvertex get rerouted to originate from t_i, while all the incoming arcs of the removed cutvertex are rerouted to terminate at t_o.

Construction of a WF-net that corresponds to a subtree in the 2-WF-tree is supported by two types of transformations, viz. refinements, of nets.

Definition 7.3 (Self-loop place refinement, Transition refinement).

- Let $N_1 = (P_1, T_1, F_1)$ be a net, $p \in P_1$ a place. A *self-loop place refinement* of p yields a net $N_2 = (P_1, T_1 \cup \{t_p\}, F_1 \cup \{(p, t_p), (t_p, p)\})$, denoted by $N_1[p]$.
- Let $N_1 = (P_1, T_1, F_1)$ be a net, $N_2 = (P_2, T_2, F_2)$ a WF-net with source i and sink o, $T_1 \cap T_2 = \emptyset$, $P_1 \cap P_2 = \emptyset$, and $t \in T_1$. A *transition refinement* of t by N_2 yields a net $N_3 = (P_3, T_3, F_3)$, denoted by $N_1[t/N_2]$, such that:
  - $P_3 = P_1 \cup (P_2 \setminus \{i, o\})$, $T_3 = (T_1 \setminus \{t\}) \cup T_2$, and
  - $F_3 = \{(x_1, x_2) \in F_1 | x_1 \neq t \wedge x_2 \neq t\} \cup \{(x_1, x_2) \in F_2 | \{x_1, x_2\} \cap \{i, o\}\} \cup \{(x_1, x_2) \in P_1 \times T_2 \mid (x_1, t) \in F_1 \wedge (i, x_2) \in F_2\} \cup \{(x_1, x_2) \in T_2 \times P_1 \mid (t, x_2) \in F_1 \wedge (x_1, o) \in F_2\}$. ⌟

A self-loop place refinement has been introduced in [95], while the concept of transition refinement is borrowed from [128, 131]. Figure 7.4(a) shows the result of the self-loop place refinement of place p_1 in WF-net $A1$ in Figure 7.3, whereas Figure 7.4(b) depicts the result of the transition refinement of t_{p_1} in the WF-net in Figure 7.4(a) by WF-net $A2$ in Figure 7.3.

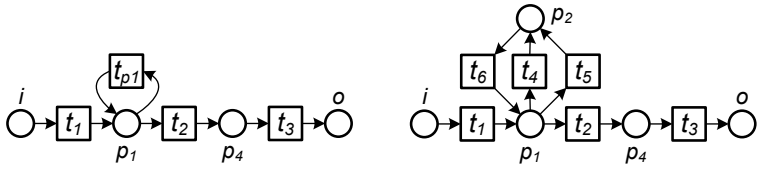


Figure 7.4. (a) A self-loop place refinement of place p_1 in WF-net $A1$ from Figure 7.3, i.e., $A1[p_1]$, and (b) a transition t_{p_1} refinement in (a) by WF-net $A2$ from Figure 7.3, i.e., $(A1[p_1])[t_{p_1}/A2]$

7.4.2. Soundness Verification Based on Biconnected Decomposition

The biconnected decomposition of a WF-net can be related to the soundness verification in two ways: First, the type of the cutvertices can be exploited to detect unsoundness. Second, we show that the biconnected decomposition can be used to modularize the soundness verification problem. The following lemma shows that a WF-net can be sound only if all the cutvertices of the corresponding short-circuit net are places.

Lemma 7.4: *Let (N, M_i), $N = (P, T, F)$, be a WF-system and N' be the short-circuit net of N. If $t \in T$ is a cutvertex of N', then (N, M_i) is unsound.* ⋆

Proof. Because t is a cutvertex of N', there exists $p' \in {\bullet}t$, $p' \neq i$, such that t is on every path $\pi_N(i, p')$. We show now by induction that t is never enabled, i.e., for every marking $M \in [N, M_i\rangle$ holds $\neg(N, M)[t\rangle$.

base: $\neg(N, M_i)[t\rangle$ as $M_i(p') = 0$, i.e., t is not enabled by the initial marking.

step: Let M' be a marking reachable from M_i by a firing sequence σ that does not contain t, i.e., t was never enabled. Let $t' \in T$ be such that $(N, M')[t'\rangle$. Assume that $t' = t$, then $M'(p') \geq 1$. If $M'(p') \geq 1$, then σ contains all the

transitions of some path $\pi_N(i, p')$ and, hence, contains t. We have reached the contradiction and, therefore, $t' \neq t$.

As t is never enabled, (N', M_i) is not live. Thus, (N, M_i) is unsound. □

According to Lemma 7.4, a transition cutvertex hints at unsoundness of the net. In case all cutvertices of a short-circuit net are places, the verification procedure can be broken down into checks of biconnected subnets of the short-circuit net. It is known that the self-loop place refinement preserves liveness, boundedness, and safeness of the net, cf., [95]. Therefore, soundness verification can be organized using the biconnected sub-WF-nets of a WF-net and the net transformations from Definition 7.3. That is, in the class of safe systems, the soundness of a system is closely related to the soundness of its biconnected sub-WF-nets.

Theorem 7.5. *Each biconnected sub-WF-net of a WF-net is safe and sound, iff the WF-net is safe and sound.* ⋆

Proof. Let N be a WF-net and let $\mathcal{T}_N^2 = (\mathcal{B}, \mathcal{C}, \rho, \eta, \triangle)$ be the 2-WF-tree of N.

($\Rightarrow$) By structural induction on the tree of the biconnected subnets.

base: If $\mathcal{T}_N^2$ contains only one biconnected subnet, i.e., $|\mathcal{B}| = 1$, then N is a biconnected WF-net. Obviously, the claim holds.

step: Let $b \in \mathcal{B}$ be a biconnected subnet. Suppose that the claim holds for all $a^{\triangle}$ such that $a \in \eta(b)$. We show by induction that the claim is also true for $b^{\triangle}$. $b^{\star}$ is a biconnected sub-WF-net of N and, hence, is safe and sound. Let $a \in \eta(b)$ and $c \in \mathcal{C}$ be such that $(b, c, a) \in \triangle$. A WF-net $b' = b^{\star}[c]$ with a self-loop transition t_c is safe and sound. A WF-net $b'[t_c/a^{\triangle}]$ is safe and sound, see statement 4 of Theorem 3 in [131]. Thus, after refining $b^{\star}$ with all the biconnected WF-nets that correspond to subnets from $\eta(b)$, one obtains a safe and sound WF-net that is equal to $b^{\triangle}$.
As $\rho^{\triangle}$ is equal to N, the claim holds.

($\Leftarrow$) The claim trivially holds by following ($\Rightarrow$) in the reverse direction. □

Therefore, it suffices to show that at least one biconnected sub-WF-net is not safe and sound in order to conclude that the WF-net is not safe and sound. As biconnected sub-WF-nets can be computed in time linear to the size of a net, the biconnected decomposition step does not add to the overall complexity of soundness verification.

7.4.3. Feedback on Unsoundness

We illustrate the feedback given by our verification approach by the exemplary model that is depicted in Figure 7.5, along with its biconnected decomposition and the biconnected sub-WF-nets. Following on the results presented in the previous section, the first check to verify soundness refers to the type of the cutvertices. A transition cutvertex hints at unsoundness of the net and, therefore, constitutes valuable diagnostic information. For our example, we see that there is one transition cutvertex, viz. transition t_1. This transition is returned to a process analyst as a cause of unsoundness of the WF-net. Apparently, this transition is

never enabled, as the marking of one of the places in its preset depends on the firing of the very same transition. Hence, resolution of unsoundness has to consider the condition for enabling of transition t_1.

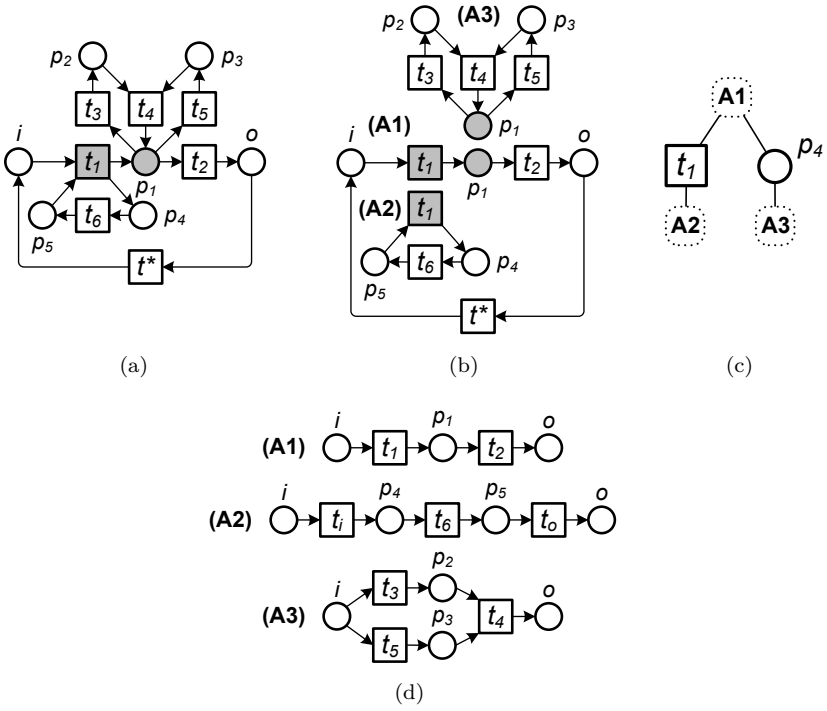


Figure 7.5. (a) A short-circuit net of an unsound WF-net, (b) biconnected subnets, (c) the 2-WF-tree, (d) the biconnected sub-WF-nets

Second, we have shown that the soundness verification procedure can be broken down into checks of biconnected subnets of the short-circuit net. Once an unsound biconnected sub-WF-net is identified, it is presented to the process analyst. Again, this constitutes valuable diagnostic information. The cause of unsoundness is localized in a certain subnet and, thus, separated from the correct remaining parts of the WF-net. Referring to our example in Figure 7.5, we see that the biconnected subnet $A3$ induced by the cutvertex p_1 is not sound. This subnet introduces an additional cause for unsoundness. The resolution of unsoundness also has to focus on this particular subnet.

7.5. The Triconnected Step

This section discusses connectivity specific aspects of the soundness verification of biconnected WF-nets. The biconnected WF-nets contain no cutvertices; they may, however, contain separation pairs that induce triconnected subnets.

7.5.1. Triconnected Decomposition of a Biconnected WF-net

The sequential algorithm for computing triconnected components in a biconnected graph proposed in [54] runs in linear time. In [48], the authors discuss implementation aspects of the algorithm. Let (V, E) be a biconnected graph, then the algorithm requires time and space proportional to $max(|V|, |E|)$. The triconnected components are used in [124] for analyzing the structure of directed graphs; the triconnected components form a hierarchy of SESE subgraphs of a directed graph. In [145, 146], the authors propose a technique, viz. the Refined Process Structure Tree (RPST), that computes a hierarchy of SESE subgraphs in the general case when the graph contains vertices with multiple incoming and multiple outgoing edges. The subgraphs of the RPST are canonical, i.e., the RPST describes all the subgraphs that do not pairwise overlap on the sets of edges. In Section 3.3, we explained the relation between the triconnected components of the normalized version of a graph, where each vertex with multiple incoming and multiple outgoing edges is split into two, and the RPST of the graph. That is, the triconnected components of a normalized graph define its RPST.

In the following, we investigate the triconnected components of the normalized biconnected WF-nets. Every net can be normalized.

Definition 7.6 (Normalized net).
Let $N = (P, T, F)$ be a net.

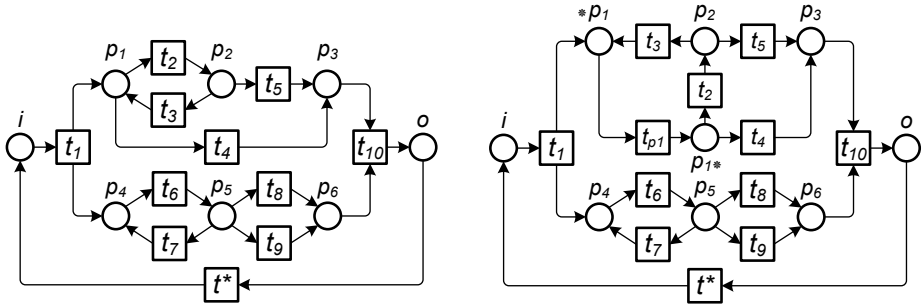
- A *splitting* of $p \in P$ is *applicable*, iff $|\bullet p| > 1 \wedge |p\bullet| > 1$. The application results in the net $N' = (P', T', F')$, where $P' = ((P \setminus p) \cup \{*p, p*\}$, $T' = T \cup \{t_p\}$, and $F' = (F \setminus \{(x_1, x_2) \in F \mid x_1 = p \vee x_2 = p\}) \cup \{(t, *p) \in T \times \{*p\} \mid (t, p) \in F\} \cup \{(p*, t) \in \{p*\} \times T \mid (p, t) \in F\} \cup \{(*p, t_p), (t_p, p*)\})$.
- A *splitting* of $t \in T$ is *applicable*, iff $|\bullet t| > 1 \wedge |t\bullet| > 1$. The application results in the net $N' = (P', T', F')$, where $P' = P \cup \{p_t\}$, $T' = ((T \setminus t) \cup \{*t, t*\}$, and $F' = (F \setminus \{(x_1, x_2) \in F \mid x_1 = t \vee x_2 = t\}) \cup \{(p, *t) \in P \times \{*t\} \mid (p, t) \in F\} \cup \{(t*, p) \in \{t*\} \times P \mid (t, p) \in F\} \cup \{(*t, p_t), (p_t, t*)\})$.
- N is *normalized*, iff N has no applicable splitting. ⌟

Application of a splitting preserves liveness, safeness, and boundedness of a net, cf., [95]. The order of splittings has no effect on the final result. Figure 7.6(a) shows the short-circuit net of a biconnected WF-net. The net has an applicable splitting of place p_1. Thus, it is not normalized. Figure 7.6(b) shows the result of splitting p_1. As there are no further splittings applicable, the net in Figure 7.6(b) is normalized, or is the *normalized version* of the net in Figure 7.6(a).

Every separation pair of the triconnected component of the normalized short-circuit net induces a canonical triconnected subnet of the corresponding WF-net, see Section 3.3.1. In the following, we adapt Definitions 3.2, 3.3, and 3.4 to nets.

Definition 7.7 (Boundary, Entry, Exit, Triconnected subnet).
Let $N = (P, T, F)$ be a biconnected WF-net with source place i and sink place o. Let $\omega = (P_\omega, T_\omega, F_\omega)$ be a subnet of N, and let $x \in P_\omega \cup T_\omega$ be a node of ω.

- x is a *boundary* node of ω, iff $\exists\ f \in in(x) \cup out(x) : f \notin F_\omega$.
- x is an *entry* of ω, iff $in(x) \cap F_\omega = \varnothing$ or $out(x) \subseteq F_\omega$.
- x is an *exit* of ω, iff $out(x) \cap F_\omega = \varnothing$ or $in(x) \subseteq F_\omega$.

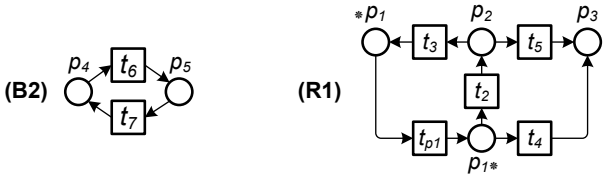


(a) (b)

Figure 7.6. (a) A short-circuit net, and (b) its normalized version

- ω is a *triconnected subnet* of N, iff ω has exactly two boundary nodes, one is an entry and the other one is an exit, denoted by $\omega_{\triangleleft}$ and $\omega_{\triangleright}$, respectively.
- ω is a *canonical* triconnected subnet in a set of all triconnected subnets Σ of N, iff $\omega \in \Sigma$ and $\forall\ \gamma = (P_\gamma, T_\gamma, F_\gamma) \in \Sigma \setminus \omega : (F_\omega \cap F_\gamma = \varnothing) \vee (F_\omega \subset F_\gamma) \vee (F_\gamma \subset F_\omega)$.⌟

Note that we speak of *PP-*, *TT-*, *PT-*, *TP-bordered* triconnected subnets, depending on the type (place or transition) of the entry and exit of the respective subnet.



(a) (b)

Figure 7.7. Triconnected subnets

Figure 7.7 shows two of many canonical triconnected subnets of the net in Figure 7.6(b). Subnet $B2$ is caused by the separation pair $\{p_4, p_5\}$, whereas subnet $R1$ is caused by $\{*p_1, p_3\}$. p_4 is the entry and p_5 is the exit of $B2$. Similarly, $*p_1$ and p_3 are the entry and exit of $R1$. Both subnets are PP-bordered.

Recall from Section 3.3 that there exist four structural classes of triconnected components and, thus, of triconnected subnets: Each flow defines a subnet of a *trivial* class, e.g., $\{(p_4, t_6)\}$ in $B2$. A subnet is a *polygon* if it decomposes into a sequence of subnets where the exit of a subnet is the entry of the next subnet in the sequence, e.g., two trivial subnets $\{(p_4, t_6)\}$ and $\{(t_6, p_5)\}$ form a polygon inside of $B2$. A subnet is a *bond* if it decomposes into a set of subnets that share boundary nodes, e.g., two polygons $\{(p_4, t_6), (t_6, p_5)\}$ and $\{(p_5, t_7), (t_7, p_4)\}$ share boundary nodes $\{p_4, p_5\}$ and form bond $B2$. If a subnet cannot be classified as trivial, polygon, or bond, it is said to be *rigid*, e.g., subnet $R1$. Note that names of subnets hint at their structural class.

Definition 7.8 (The tree of the triconnected subnets).
Let $N = (P, T, F)$ be a normalized biconnected WF-net and let N' be its short-circuit net with the extra transition t^*. The *tree of the triconnected subnets*, or the *3-WF-tree*, of N is a tuple $\mathcal{T}_N^3 = (\Omega, \psi, \chi, \tau)$, where:

- Ω is the set of all canonical triconnected subnets of N',
- $\psi = (P_\psi, T_\psi, F_\psi) \in \Omega$ is the *root* of $\mathcal{T}_N^3$, iff $\nexists\ \omega = (P_\omega, T_\omega, F_\omega) \in \Omega : F_\psi \subset F_\omega$,
- $\chi : \Omega \to \mathcal{P}(\Omega)$ assigns to each triconnected subnet its child triconnected subnets; for subnet $\omega_1 \in \Omega$, ω_1 is a *parent* of $\omega_2 \in \Omega$ and ω_2 is a *child* of ω_1, iff $\omega_2 \in \chi(\omega_1)$,
- $\tau : \Omega \to \{trivial, polygon, bond, rigid\}$ assigns to each triconnected subnet its class. ⌟

The tree of the triconnected subnets of a biconnected WF-net is defined by the RPST of its normalized short-circuit net. Trivial subnets are leaves in 3-WF-trees. Any polygon containing other polygons or bond containing other bonds are recognized as a single polygon or bond, respectively. For our purposes, we further classify polygon and bond subnets: A subnet $\omega \in \Omega$ is a *simple* polygon, iff $\tau(\omega) = polygon$ and $\forall\ \alpha \in \chi(\omega) : \tau(\alpha) = trivial$. A subnet $\omega \in \Omega$ is a *sequence*, iff the subnet is a simple polygon or trivial. Let $h : \Omega \to \mathbb{N}_0$ assign heights to triconnected subnets, i.e., the length of the longest downward path from the corresponding node in the 3-WF-tree to a node that corresponds to a trivial subnet. A subnet $\omega \in \Omega$ is a *simple* bond, iff $\tau(\omega) = bond$ and $h(\omega) \leq 2$. Finally, a bond $\omega \in \Omega$ is a *loop*, iff there exists $\gamma \in \chi(\omega)$ such that $\gamma_\triangleright = \omega_\triangleleft$.

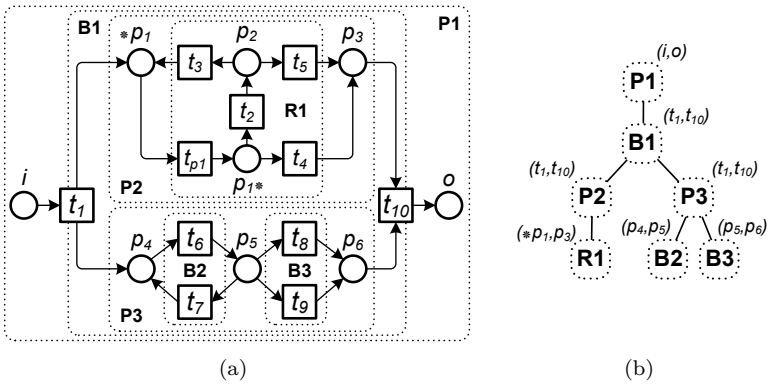


Figure 7.8. (a) A normalized net and its triconnected subnets, and (b) the 3-WF-tree

Figure 7.8 exemplifies the 3-WF-tree: Figure 7.8(a) shows the net in Figure 7.6(b) along with its canonical triconnected subnets. Each triconnected subnet is defined by the flow that is inside or intersects the corresponding box with the dotted line. Figure 7.8(b) gives an alternative representation of the 3-WF-tree, i.e., a tree specified by the containment hierarchy of subnets. Each node in the tree is annotated with the pair of nodes of the net, where the first node is the entry and the second node is the exit of the corresponding subnet.

The 3-WF-tree specifies a compositional structure of the net. The net in Figure 7.8(a) is composed of the top level polygon $P1$ with entry i and exit o. Polygon $P1$ contains bond $B1$ with entry t_1 and exit t_{10}. Observe that $P1$ also contains two trivial subnets: $\{(i,t_1)\}$ and $\{(t_{10},o)\}$. Note that for simplicity reasons, we do not explicitly visualize sequence subnets. Bond $B1$ contains two polygons that share boundary nodes t_1 (an entry) and t_{10} (an exit) with the bond, where $P2$ corresponds to the upper branch and $P3$ corresponds to the lower branch. Bonds $B2$ and $B3$ are simple bonds, both composed of two sequences. Moreover, bond $B2$ is also a loop. The net contains one rigid subnet – $R1$.

A triconnected subnet contains all the triconnected subnets, as well as the behavior that they specify, downwards in the hierarchy of the 3-WF-tree. However, the behavior induced by a triconnected subnet can be made explicit by abstracting the behavior of all its descendant subnets.

Definition 7.9 (Abstraction).
Let $N = (P,T,F)$ be a normalized biconnected WF-net. Let $\mathcal{T}_N^3 = (\Omega,\psi,\chi,\tau)$ be its 3-WF-tree, $\omega = (P_\omega,T_\omega,F_\omega) \in \Omega$, and $\gamma = (P_\gamma,T_\gamma,F_\gamma) \in \chi(\omega)$. An *abstraction* of γ in ω by transition t_γ results in a subnet $\omega_\gamma^\star$, such that:

- If $\gamma_\triangleleft \in P \wedge \gamma_\triangleright \in P$, then $\omega_\gamma^\star = (P_\omega \setminus (P_\gamma \setminus \{\gamma_\triangleleft,\gamma_\triangleright\}), (T_\omega \setminus T_\gamma) \cup \{t^\gamma\}, (F_\omega \setminus F_\gamma) \cup \{(\gamma_\triangleleft,t^\gamma),(t^\gamma,\gamma_\triangleright)\})$.
- If $\gamma_\triangleleft \in P \wedge \gamma_\triangleright \in T$, then $\omega_\gamma^\star = ((P_\omega \setminus (P_\gamma \setminus \{\gamma_\triangleleft\})) \cup \{p^\gamma\}, (T_\omega \setminus (T_\gamma \setminus \{\gamma_\triangleright\})) \cup \{t^\gamma\}, (F_\omega \setminus F_\gamma) \cup \{(\gamma_\triangleleft,t^\gamma),(t^\gamma,p^\gamma),(p^\gamma,\gamma_\triangleright)\})$.
- If $\gamma_\triangleleft \in T \wedge \gamma_\triangleright \in P$, then $\omega_\gamma^\star = ((P_\omega \setminus (P_\gamma \setminus \{\gamma_\triangleright\})) \cup \{p^\gamma\}, (T_\omega \setminus (T_\gamma \setminus \{\gamma_\triangleleft\})) \cup \{t^\gamma\}, (F_\omega \setminus F_\gamma) \cup \{(\gamma_\triangleleft,p^\gamma),(p^\gamma,t^\gamma),(t^\gamma,\gamma_\triangleright)\})$.
- If $\gamma_\triangleleft \in T \wedge \gamma_\triangleright \in T$, then $\omega_\gamma^\star = ((P_\omega \setminus P_\gamma) \cup \{p^{*\gamma},p^{\gamma *}\}, (T_\omega \setminus (T_\gamma \setminus \{\gamma_\triangleleft,\gamma_\triangleright\})) \cup \{t^\gamma\}, (F_\omega \setminus F_\gamma) \cup \{(\gamma_\triangleleft,p^{*\gamma}),(p^{*\gamma},t^\gamma),(t^\gamma,p^{\gamma *}),(p^{\gamma *},\gamma_\triangleright)\})$. ⌟

An abstraction of a child subnet results in a net where the child subnet is replaced by a fresh transition. A *triconnected (sub-)WF-net* of N, denoted by ω^ι, can be obtained from ω by abstracting all its non-trivial child subnets, introducing a path from a fresh source place i leading to $\omega_\triangleleft$, possibly through a transition, and introducing a path from $\omega_\triangleright$ to a fresh sink place o, possibly through a transition.

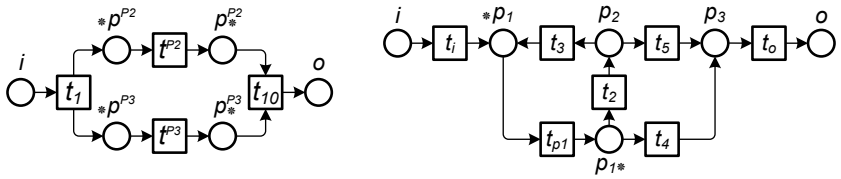


Figure 7.9. Triconnected sub-WF-nets: (a) B1, and (b) R1

Figure 7.9 shows two triconnected sub-WF-nets that correspond to subnets $B1$ and $R1$ of the net in Figure 7.8(a). Note that the time to compute and space to store all the triconnected sub-WF-nets of a biconnected WF-net is linear to the size of the net.

7.5.2. Soundness Verification Based on Triconnected Decomposition

In the following, we discuss how the triconnected sub(-WF-)nets can be used to judge if a net behaves correctly. Similar to Section 7.4.2, we start by characterizing separating sets of a net. However, in this case the sets are separation pairs. First, we identify simple bonds that hint at a WF-net being unsound.

Lemma 7.10: *[Pruning] Let (N, M_i), $N = (P, T, F)$, be a biconnected WF-system. Let $\mathcal{T}_N^3 = (\Omega, \psi, \chi, \tau)$ be its 3-WF-tree, with a simple bond $\omega \in \Omega$. If ω is neither PP-, nor non-loop TT-, nor non-loop TP-bordered, then (N, M_i) is unsound.* ⋆

Proof. Because ω is a simple bond, it holds that all child subnets of ω are sequences. There exist eight classes of simple bonds that are based on two criteria: is the bond a loop (ii) or not (i), and is the bond (a) PP-, (b) PT-, (c) TP-, or (d) TT-bordered. Figure 7.10 visualizes all eight classes of simple bonds. In the figure, each directed arc stands for $n \in \mathbb{N}$ sequence subnets with entries and exits that correspond to the source and target of the arc. Note that we always assume the left node of a bond to be its entry node. Then, it trivially holds: If ω is of class (i.b), (ii.c), or (ii.d), then there is a dead transition in the WF-system. If ω is of class (ii.b), then there is a live-lock in the WF-system. In both cases, the WF-system is not sound. Moreover, if at least one child subnet of ω is trivial, then ω cannot be of class (i.a), (i.d), (ii.a), and (ii.d), due to the bipartite property of Petri nets. □

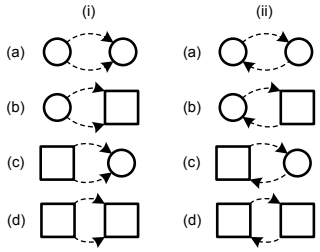


Figure 7.10. Simple bond classes

A biconnected WF-system is said to be *pruned*, if it is not identified by Lemma 7.10 as unsound. Pruning is applicable for general biconnected WF-systems, but considers solely simple bonds. For the class of biconnected *free-choice* WF-nets, we can even provide a complete characterization of bonds. This is due to the fact that the free-choice property implies a tight coupling of the syntax and semantics for sound WF-systems, cf., [131, 68]. In order to present the results, we recall some definitions from [39].

Definition 7.11 (Circuit, Handle, Bridge).
Let $N = (P, T, F)$ be a net.

- A path $\pi_N(x_1, x_k)$ of N is a *circuit*, iff $(x_k, x_1) \in F$ and no node occurs more than once in the path.
- For a partial subnet $N' = (P', T', F')$ of N, a path $\pi_N(x_1, x_k)$ (where all x_i are distinct) of N is a *handle* of N', iff $\pi_N \cap (P' \cup T') = \{x_1, x_k\}$.

- For two partial subnets $N' = (P', T', F')$, $N'' = (P'', T'', F'')$ of N, a path $\pi_N(x_1, x_k)$ (where all x_i are distinct) of N is a *bridge* from N' to N'', iff $\pi_N \cap (P' \cup T') = \{x_1\}$ and $\pi_N \cap (P'' \cup T'') = \{x_k\}$. ⌟

Note that we speak of PP-, TT-, PT-, TP- handles and bridges, depending on the type (place or transition) of the start and the end node of the corresponding path. Finally, the following two lemmas show that a bond of a sound biconnected free-choice WF-system is either place or transition bordered, and that each loop is place bordered.

Lemma 7.12: *Let (N, M_i), $N = (P, T, F)$, be a biconnected free-choice WF-system. Let $\mathcal{T}_N^3 = (\Omega, \psi, \chi, \tau)$ be its 3-WF-tree, with a bond $\omega \in \Omega$. If ω is TP-bordered or PT-bordered, then (N, M_i) is unsound.* ⋆

Proof. Let N' be the short-circuit net of N. Assume ω is a bond with $\{p, t\}$ boundary nodes, $p \in P$ and $t \in T$. There exists a circuit Γ in N' that contains $\{p, t\}$. Let Γ_ω be a subpath of Γ inside ω. There exists a child subnet γ of ω that contains Γ_ω. A bond has at least two child subnets. Let ϑ be a child of ω, $\vartheta \neq \gamma$. We distinguish two cases:

- Let H be a path from p to t contained in ϑ. H is a PT-handle of Γ. In a live and bounded free-choice system, H is bridged to Γ_ω through a TP-bridge K, see Proposition 4.2 in [39]. This implies that $\vartheta = \gamma$; otherwise bond ω contains path K that is not inside of its single child subnet.
- Let H be a path from t to p contained in ϑ. H is a TP-handle of Γ. In a live and bounded free-choice system, no circuit has TP-handles, see Proposition 4.1 in [39]. Thus, (N, M_i) is not a sound free-choice WF-system.

Therefore, ω either has a single child subnet, in which case ω is not a bond, or (N, M_i) is not a sound free-choice WF-system. □

Lemma 7.13: *Let (N, M_i), $N = (P, T, F)$, be a biconnected free-choice WF-system. Let $\mathcal{T}_N^3 = (\Omega, \psi, \chi, \tau)$ be its 3-WF-tree, with a loop $\omega \in \Omega$. If ω is TP-bordered, TT-bordered, or PT-bordered, then (N, M_i) is unsound.* ⋆

Proof. Because of Lemma 7.12, ω is either place or transition bordered. Assume ω is transition bordered. There exists a place p such that $p \in {}^\bullet\omega_\triangleleft \cap P_\omega$, $M_i(p) = 0$. Transition $\omega_\triangleleft$ is enabled if there exists a marking $M \in [(N, M_i)\rangle$ with $M(p) > 0$. Since ω is a connected subnet, for all $t \in T_\omega \setminus \{\omega_\triangleleft, \omega_\triangleright\}$ all edges are in ω, i.e., $(in(t) \cup out(t)) \subseteq F_\omega$. Thus, every path from i to p visits $\omega_\triangleleft$. Thus, $M(p) > 0$ if $\omega_\triangleleft$ has fired, was enabled before. We reached a contradiction. Transition $\omega_\triangleleft$ is never enabled and N is not live, and hence, not sound. Since any loop is not transition bordered, it is place bordered. □

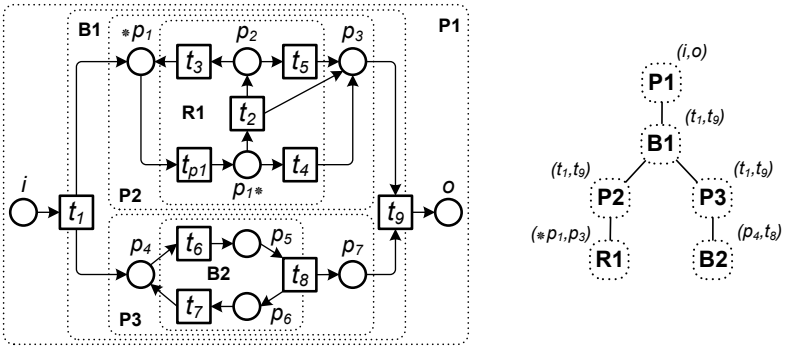
Finally, similar as in Section 7.4.2, we propose to organize the soundness verification of biconnected WF-systems from individual checks of its sub-WF-nets, viz. triconnected sub-WF-nets.

Theorem 7.14. *Each triconnected sub-WF-net of a biconnected WF-net is safe and sound, iff the WF-net is safe and sound.* ⋆

Proof. By structural induction on the tree of the triconnected subnets; analogous to the proof of Theorem 7.5. □

It suffices to show that at least one of the triconnected sub-WF-nets of a biconnected WF-net is not safe and sound in order to conclude that the net is not safe and sound. The analysis discussed in this section can be performed in the time linear to the size of a net and, hence, does not influence the overall complexity of soundness verification.

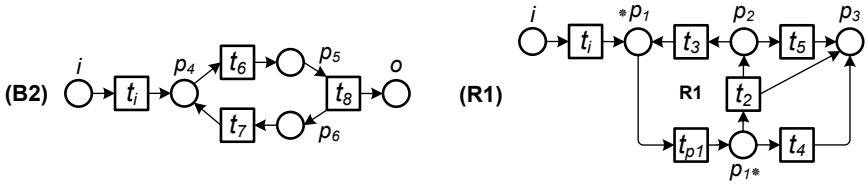
7.5.3. Feedback on Unsoundness



(a) (b)

Figure 7.11. (a) A short-circuit net of an unsound biconnected WF-net, (b) the 3-WF-tree

We illustrate the feedback given by our approach for the exemplary workflow net depicted together with its 3-WF-tree in Figure 7.11. The workflow net is clearly not sound. A first cause of unsoundness can already be identified in the pruning step. Bond $B2$ caused by the separation pair $\{p_4, t_8\}$ is a simple bond; all its child subnets are sequences. Furthermore, this simple bond is PT-bordered. This, in turn, contradicts with Lemma 7.10. Hence, the respective separating nodes, i.e., place p_4 and transition t_8, are returned to a process analyst as diagnostic information. Figure 7.12(a) shows the triconnected sub-WF-net $B2$.



(a) (b)

Figure 7.12. Triconnected sub-WF-nets: (a) B2, and (b) R1

Further on, the example given in Figure 7.11 also illustrates the kind of feedback given once an unsound triconnected subnet is identified. Here, the triconnected sub-WF-net $R1$ induced by the separation pair $\{*p_1, p_3\}$ is not sound, see Figure 7.12(b),

which according to Theorem 7.14 makes the whole net unsound. Observe that the net in Figure 7.11(a) is free-choice. Therefore, the triconnected sub-WF-net $R1$ is an additional cause for unsoundness and has to be corrected in order to ensure soundness.

7.6. The 4-Connected Step

This section is devoted to soundness verification of rigid (triconnected sub-)WF-nets. The short-circuit net of a rigid WF-net contains no separation pairs. Still, the net might contain separating sets composed of three nodes, which induce 4-connected components. Those might be leveraged for soundness verification.

7.6.1. 4-Connected Decomposition of a Triconnected WF-net

This section explains an approach for the discovery of the 4-connected subnets in a triconnected WF-net. The authors are not aware of any existing dedicated algorithm that can be directly applied. However, the discovery can be organized by employing the technique from the triconnected step, see Section 7.5.1. The 4-connected subnets of a triconnected WF-net are detected as follows: First, a node is removed from the net. This step is referred to as *one-step connectivity reduction*. Afterwards, separation pairs of the modified net are discovered by constructing its 3-WF-tree. The removed node and each separation pair captured in the tree form a separating set composed of three nodes. The procedure should be repeated for each node in the net.

Definition 7.15 (One-step connectivity reduction).
Let $N = (P, T, F)$ be a rigid triconnected WF-net with source i and sink o.

- A node $x \in P \cup T$ is a *separation node*, iff $|\bullet x| + |x \bullet| > 2$.
- Given a separation node $x \in P \cup T$, a node $y \in P \cup T$ is in the set of *reduced nodes* $\tilde{x}$, iff $y = x$ or there is a path $\pi_N(x, y)$ or a path $\pi_N(y, x)$ that does not contain a separation node of N other than x.
- Given a separation node $x \in P \cup T$, the *one-step connectivity reduction* of N by x yields a subnet $N_x = (P_x, T_x, F_x)$ induced by nodes $P_x = P \setminus \tilde{x}$ and $T_x = T \setminus \tilde{x}$.
- A subnet N_x derived via one-step connectivity reduction of N by x is *well-reduced*, iff its short-circuit net N'_x is strongly connected. ⌟

The one-step connectivity reduction is illustrated in Figure 7.13. Figure 7.13(a) depicts the short-circuit net of a rigid WF-net, while Figure 7.13(b) shows the short-circuit of a subnet derived via one-step connectivity reduction by separation node p_2. As the resulting net is well-reduced, its 3-WF-tree is computed, see Figure 7.13(c).

The well-reduced subnets derived via one-step connectivity reduction can be related to the structure of the original net.

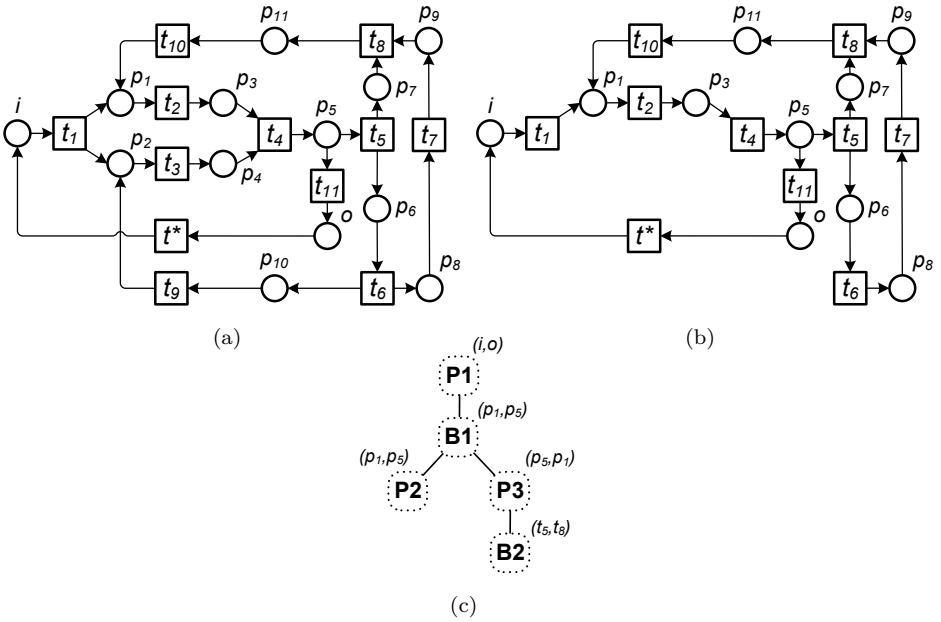


Figure 7.13. (a) A short-circuit net of a rigid triconnected WF-net, (b) an exemplary one-step connectivity reduction of (a), and (c) the 3-WF-tree of (b)

Proposition 7.1: *Let N be a rigid WF-net, N' its short-circuit net, and N_x a well-reduced subnet of N' derived via one-step connectivity reduction. Each bond subnet of N_x contains a handle for a circuit in N'.* ⋆

Proof. Let $\omega = (P_\omega, T_\omega, F_\omega)$ be a bond subnet of N_x. As N_x is well-reduced, its short-circuit net N'_x is strongly connected. Hence, N'_x has a circuit that contains $\omega_\triangleleft$ and $\omega_\triangleright$. As ω is of type bond, it has at least two node-disjoint paths between $\omega_\triangleleft$ and $\omega_\triangleright$, where *node-disjoint* paths are paths with only end nodes in common. One of these paths is a subpath for the circuit in N'_x, whereas the second one is its handle. It trivially holds that the circuit is also the circuit in N'. □

7.6.2. Soundness Verification Based on 4-Connected Decomposition

For the class of free-choice nets, the 4-connected decomposition can be used to check mandatory conditions for the soundness of a rigid triconnected WF-net. Soundness of a free-choice WF-net can be decided using the *Rank Theorem* [22], which relates the well-formedness property (there exists a live and bounded marking for the net) to the incidence matrix of the net. According to [66], well-formedness and, thus, soundness of a free-choice net can be decided in $O(|P|^2 \cdot |T|)$ time. Still, the 4-connected decomposition allows for more efficient checks of mandatory conditions for soundness. Consequently, these checks might be applied before soundness is assessed based on the Rank Theorem. The following lemma provides such a check by partial structural characterizations of sound nets.

Lemma 7.16: *Let (N, M_i) be a rigid free-choice WF-system and N_x a well-reduced subnet of N derived via one-step connectivity reduction. If N_x contains a bond that is non-loop TP-bordered or loop PT-bordered, then (N, M_i) is unsound.* ⋆

Proof. Let $\omega = (P_\omega, T_\omega, F_\omega)$ be a bond subnet of N_x. According to Proposition 7.1, ω induces a handle for a circuit in N. We distinguish two cases:

- ω is non-loop TP-bordered. Then, the handle induced by ω is a TP-handle.
- ω is loop PT-bordered. For $\omega_\triangleleft$ we know that either $in(\omega_\triangleleft) \cap F_\omega = \varnothing$ or $out(\omega_\triangleleft) \subseteq F_\omega$. The former is not possible as ω is a loop. From the latter, it follows that there is only one path $\pi_N(\omega_\triangleleft, \omega_\triangleright)$ in ω. Thus, the handle induced by ω for a circuit in N (Proposition 7.1) is a path $\pi_N(\omega_\triangleright, \omega_\triangleleft)$, i.e., a path from the exit to the entry of ω. As ω is PT-bordered, this path is a TP-handle.

In both cases, ω induces a TP-handle for a circuit in N. According to Proposition 4.1 in [39], no circuit of a live and bounded free-choice net has TP-handles.□

In the same vein, an acyclic rigid subnet has to meet a structural requirement in order to be sound. The following lemma applies to all free-choice biconnected WF-nets and not only to those derived via one-step connectivity reduction from a triconnected WF-net.

Lemma 7.17: *Let (N, M_i) be a biconnected free-choice WF-system. If N contains an acyclic rigid subnet that is TP-bordered, then (N, M_i) is not sound.* ⋆

Proof. Let $\omega = (P_\omega, T_\omega, F_\omega)$ be a rigid subnet of N that is TP-bordered. Then, there is a circuit in the short-circuit net N' that contains both boundary nodes, $\omega_\triangleleft \in T$ and $\omega_\triangleright \in P$. According to Menger's theorem, cf., [87, 122], there are two node-disjoint undirected paths from $\omega_\triangleleft$ to $\omega_\triangleright$. As ω is acyclic, all edges on these paths are directed equally, yielding two paths $\pi^1_N(\omega_\triangleleft, \omega_\triangleright)$ and $\pi^2_N(\omega_\triangleleft, \omega_\triangleright)$. Thus, one of these paths is a TP-handle of the subnet containing the circuit. According to Proposition 4.1 in [39], no circuit of a live and bounded free-choice net has TP-handles. □

The application of the 4-connected decomposition allows for checking the mandatory conditions for soundness of a rigid free-choice WF-net in $O(|P \cup T|^2)$ time. For each separation node in a rigid free-choice WF-net, we apply a one-step connectivity reduction and derive the respective 4-connected components in linear time. For these components, Lemmas 7.16 and 7.17 can be checked in linear time as well. Consequently, this technique can be seen as a preliminary step that is applied before the more costly assessment of soundness based on the Rank Theorem.

7.6.3. Feedback on Unsoundness

We illustrate the feedback given in case unsoundness is detected by 4-connected decomposition with an example. Figure 7.14 depicts the short-circuit net of a rigid triconnected WF-net similar to the one introduced above in Figure 7.13. However, the net depicted in Figure 7.14(a) is not sound. The cause for unsoundness is detected as follows: Applying the one-step connectivity reduction by separation

node p_2 leads to the well-reduced WF-net depicted in Figure 7.14(b). For this subnet, in turn, the triconnected decomposition is applied, which results in the 3-WF-tree illustrated in 7.14(c). The 3-WF-tree reveals that bond $B2$ is non-loop TP-bordered. According to Lemma 7.16, such a bond violates a mandatory condition for soundness of the WF-net. As diagnostic information, the respective handle induced by this bond is presented to the process analyst. That is, the two paths, one directly between t_5 and p_7 nodes and one via the nodes p_6, t_6, p_8, and t_7, identify the cause of the unsoundness.

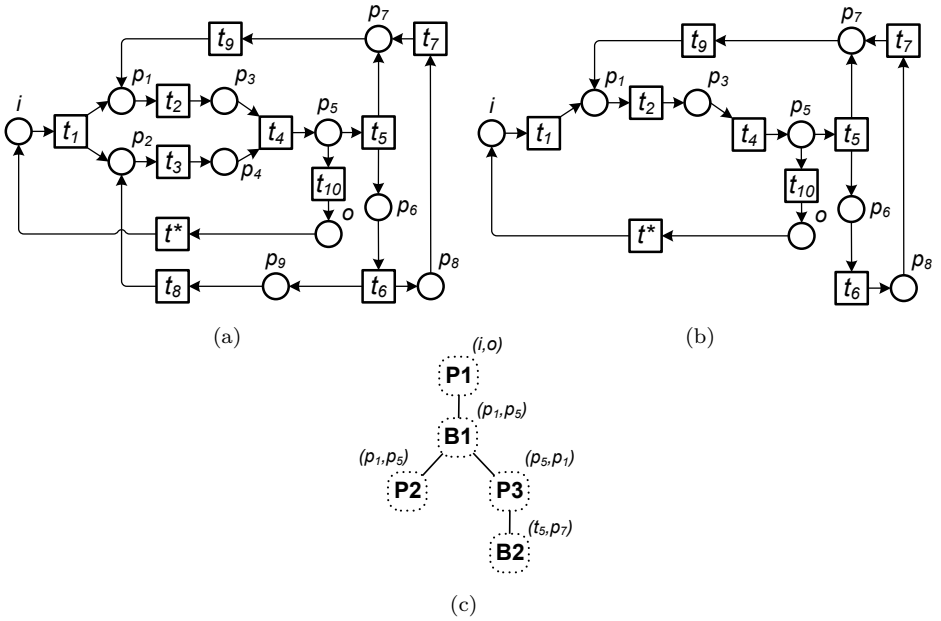


Figure 7.14. (a) An unsound short-circuit net of a rigid triconnected WF-net, (b) an exemplary one-step connectivity reduction of (a), and (c) the 3-WF-tree of (b)

Feedback on unsoundness detected based on Lemma 7.17 can be provided in a similar way. The entry and exit of an acyclic TP-bordered rigid subnet are cause of unsoundness which must be corrected in order to ensure soundness. In fact, the WF-net in Figure 7.14(a) contains a single TP-bordered rigid subnet with entry t_1 and exit p_5. However, the subnet is cyclic and, thus, in this particular case does *not* hint at unsoundness of the WF-net.

7.7. Application

This section shows how the results presented above can be applied for the purpose of soundness verification. Algorithm 8 formalizes the verification procedure that is obtained by integrating the individual results proposed in Sections 7.4 to 7.6. The algorithm specifies a predicate which takes a WF-net as input and returns `true`, if the net is sound, and `false`, if the net is unsound. Note that Algorithm 8 does not

formalize the way feedback is given in case of unsoundness; this was suppressed in order to keep the formalization concise. As discussed above, feedback can directly be given in terms of the elements that violate one of the necessary conditions for soundness.

Algorithm 8: Connectivity-based soundness verification of a WF-net

Input: N – a WF-net
Output: *true* if N is sound; *false* otherwise

```
1  N' = (P', T', F') := Get short-circuit net of N
2  T^2_{N'} = (B, C, ρ, η, △) := Get the tree of the biconnected subnets of N'
3  foreach c ∈ C do if c ∈ T' then return false
4  areFC := areSound := true
5  foreach b ∈ B do
6  |  b* := Get normalized biconnected sub-WF-net of b
7  |  isFC := true if b* is free-choice; false otherwise
8  |  areFC := areFC ∧ isFC
9  |  T^3_{b*} = (Ω, ψ, χ, τ) := Get the tree of the triconnected subnets of b*
10 |  foreach ω ∈ Ω do
11 |  |  if (ω is simple bond) ∧ (ω is PT-, loop TP-, or loop TT-bordered)
   |  |  then return false
12 |  |  if areFC ∧ (ω is bond) ∧ (ω is PT-, TP-, or loop TP-bordered)
   |  |  then return false
13 |  |  if areFC ∧ (ω is rigid) then
14 |  |  |  if (ω is acyclic) ∧ (ω is TP-bordered) then return false
15 |  |  |  foreach ω' gained from ω by one-step connectivity reduction do
16 |  |  |  |  if ω' has bond that is non-loop TP-, or loop PT-bordered
   |  |  |  |  then return false
17 |  |  |  ω² := Get triconnected sub-WF-net of ω
18 |  |  |  isSound := true if ω² is sound (Rank Theorem); false otherwise
19 |  |  |  areSound := areSound ∧ isSound
20 if areFC ∧ areSound then
21 |  return true
22 else Perform reachability graph analysis
```

First and foremost, Algorithm 8 comprises the biconnected decomposition of the given WF-net along with the verification of the types of the respective cutvertices. Afterwards, each biconnected sub-WF-net is decomposed into its triconnected subnets. For those, the algorithm checks the necessary conditions for soundness in the general case (types of boundary nodes of simple bonds). If these conditions are met, further necessary conditions are checked if the respective net is free-choice. This comprises checks for free-choice bonds, acyclic rigid subnets, and bonds derived from rigid subnets via one-step connectivity reduction. Note that if all these conditions are met but there are sub-WF-nets which cannot be handled by the proposed theory, we cannot conclude on soundness directly. As our structural

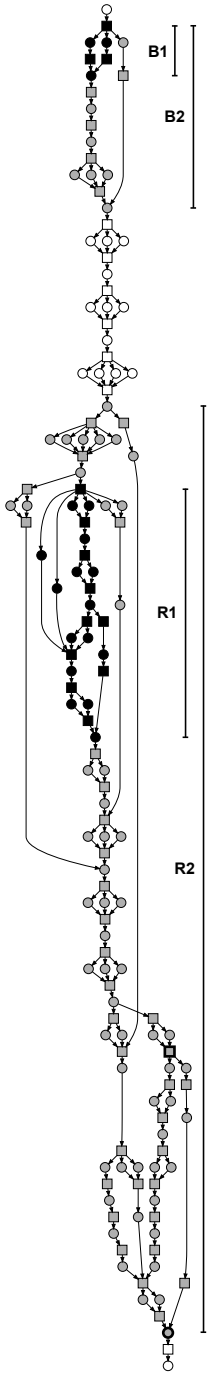


Figure 7.15. A WF-net

approach is not exhaustive, we have to incorporate additional techniques which are capable of delivering results in the general case. For the class of free-choice nets we rely on the soundness verification based on the Rank Theorem, see Section 7.6.2, whereas in the general case we can get as exhaustive as performing reachability analysis of a net, see Algorithm 8 (line 22). However, we see that for certain classes of nets, i.e., nets that are free-choice and free of triconnected rigid subnets, our checks are even sufficient.

For validation purposes, we have implemented the verification approach that is proposed in Algorithm 8 and tested it against a collection of industry process models; we have used a model collection that was first used for the soundness verification in [41]. The model collection comprises 732 WF-nets, 375 of which are sound and 357 are unsound. The authors of [41] proposed soundness checking based on heuristics and state space exploration for the triconnected decomposition of (free-choice) process graphs and also report on findings regarding the application of other verification techniques, such as model checking and coverability analysis. While the authors of [41] show impressive results in terms of efficiency, the results have been achieved under optimizations for free-choice models. Here, our focus is different, as we aim at the verification of a general class of process models at the expense of completeness of the verification.

When testing the aforementioned collection of WF-nets for soundness, we observed the following results: 732 WF-nets contain 82 108 trivial, 26 818 polygon, 9 216 bond, and 553 rigid subnets. 256 WF-nets are classified as sound. 138 bond subnets are the cause of unsoundness (see Lemmas 7.10, 7.12, and 7.13). 56 rigid subnets are identified as the cause of unsoundness via one-step connectivity reduction (see Lemma 7.16), whereas 15 rigid subnets are classified as acyclic TP-bordered (see Lemma 7.17). Moreover, we have implemented heuristics for the soundness verification that are discussed in [147]. This allowed us to additionally classify 29 rigid subnets as the cause of unsoundness and 78 rigid subnets as ones that can be present in sound WF-nets. These results show that a large share of models can be indeed verified by employing structural analysis only, among others

the analysis proposed in this article, avoiding costly state space explorations. All the checks are performed within few milliseconds, which is comparable with the results reported in [41].

To illustrate the application of Algorithm 8, we refer to a fragment of a WF-net from the model collection; the fragment is shown in Figure 7.15. The algorithm classifies the net as unsound. According to the theory of this chapter, there are several causes of unsoundness that can be discovered in the fragment. We highlight them in the figure. The net contains two TP-bordered bonds $B1$ and $B2$ (see Lemma 7.12), which are highlighted with black and, respectively, grey background at the top of the figure. Moreover, the net contains TP-bordered acyclic rigid $R1$ (see Lemma 7.17), see highlighted with black background in the middle of Figure 7.15. Finally, rigid $R2$ is also the cause of unsoundness of the net, see highlighted with grey background in the lower part of the fragment. The one-step connectivity reduction of $R2$ reveals a TP-bordered bond (see Lemma 7.16); the entry and exit nodes of the bond are depicted with a thick borderline in the lower part of Figure 7.15. Subnets, entries and exits of subnets, bonds derived via one-step connectivity reduction, these are examples of structural information which can be reported to a process analysts as a feedback on unsoundness of the WF-net.

7.8. Related Work and Conclusion

The verification approach proposed in this chapter relates to other works on the verification of behavioral models.

- In [147], the authors propose to organize the verification of workflow graphs from fragments that have a single-entry edge and a single-exit edge, i.e., $(0, 2)$-connected subgraphs in our classification. Albeit related, this work is based on the property of edge-connectivity, whereas our work leverages node-connectivity, yielding a more fine granular decomposition. Soundness checking based on heuristics and state space exploration for a triconnected decomposition, or $(2, 0)$-connected subgraphs, of a (free-choice) process graph has been proposed in [41]. Our approach goes beyond this work by embedding the idea of the triconnected soundness checking into a decomposition approach that allows for stepwise verification. In addition, we base our findings on Petri nets as a generic behavioral model. Thus, our approach is able to cope with the whole spectrum of constructs of common modeling languages that can be mapped to Petri nets (e.g., exception handling in BPEL processes), whereas the existing approaches rely on the specific notion of a process graph. Note that [41] also reports findings on the application of other verification approaches, such as LoLA and Woflan. LoLA is a tool that is capable of checking various properties of a net by inspecting its state space [158]. To increase efficiency, LoLA incorporates several techniques for state space reduction. For the investigations in [41], the authors employed CTL model checking and partial order reduction of LoLA. Woflan is a tool for verifying the soundness of workflow nets [150]. Woflan combines structural

Petri net reductions, S-coverability analysis, and state space exploration based on coverability trees into the unique verification approach. Our approach can be used in combination with these, or any other verification techniques, to deliver the *divide and conquer* strategy for verification of behavioral models.

- Verification of Petri nets can benefit from structural reductions. Reduction rules are designed to transform a net into a smaller net while preserving essential properties of the net. Reduction rules can be applied before verification to decrease effects of state space explosion. Berthelot proposed a set of rules which reduce live and bounded marked graphs to a single transition [8, 9]. Desel and Esparza, in [22], proposed a complete kit of reduction rules for free-choice Petri nets. In [95], Murata presented reduction rules which preserve the liveness, safeness, and boundedness properties of ordinary nets. Reduction rules constitute a line of active research not only for Petri nets, but also for models of concurrency which extend Petri nets. The extensions aim at support of features included in process definition languages such as BPMN, EPC, and UML activity diagrams. In [161], Wynn et al. proposed soundness-preserving reduction rules for reset WF-nets. Reset nets extend Petri nets with the concept of a reset flow. The semantics of a reset arc that connects a place and a transition is to remove all tokens from the place when the transition fires. Reset flow can be used to model cancellation. The proposed reduction rules are based on the rules for general nets and include additional restrictions with respect to reset arcs. By employing formal mappings of YAWL nets to reset nets, the same authors presented soundness-preserving reduction rules for YAWL workflow nets with cancellation regions and OR-joins [160]. Based on these rules, the authors devised an approach to check structural properties of YAWL nets, such as the weak soundness property, the soundness property, reducible cancellation regions, and convertible OR-joins. Finally, in [149], the authors defined a set of reduction rules which preserve the liveness and boundedness of reset/inhibitor nets. Reset/inhibitor nets are reset nets with the concept of an inhibitor flow. An inhibitor arc from a place to a transition can prevent the transition from being enabled if the place contains a token. Inhibitor arcs are useful when modeling blocking.
- Further related work comprises inheritance preserving transformation rules for WF-systems defined by Wil van der Aalst and Twan Basten [132]. The rules are designed to avoid problems when migrating old workflow systems to new ones. When employed, the rules restrict changes in such a way that new workflow systems inherit certain properties of the old workflow systems. The rules can be applied in two directions, i.e., to reduce or to specify a workflow system. The transformations guarantee the preservation of the soundness property. Albeit similar, these transformation rules are different from the approach proposed in this chapter. As our focus is solely on the verification of workflow systems, our structural implications on the soundness property are more general. For instance, our approach is not restricted by the class of safe workflow systems but is applicable for general

workflow nets. Moreover, we additionally describe generic structural patterns which hint at unsoundness of workflow systems, e.g., a transition cutvertex, see Lemma 7.4. These patterns allow us to formulate feedbacks in cases of unsoundness. Finally, the particular feature of our approach is that it includes instructions for identification of *all* needful (vertex connectivity-based) structural patterns within a workflow system, i.e., patterns which can confirm or reject soundness.

- In [131], Wil van der Aalst exploited the hierarchical concept of transition refinement for checking soundness of workflow systems. Similarly, the approach reported in this chapter deals with the modular analysis of the soundness property. In particular, we have investigated an efficient way for modularization of workflow nets for the purpose of their further verification. During our investigations we reused some of the results on compositionality of workflow nets which were reported in [131].

In this chapter, we have investigated the relation between the connectivity property of a workflow net and its behavioral correctness in terms of the soundness property. We showed that the soundness verification can be conducted following on a stepwise decomposition of a workflow net. For all the decomposition steps, we presented the necessary structural conditions for detecting unsoundness. We showed how our results are applied as a part of a verification procedure and tested these results against a collection of industry process models. During our tests, we observed the application of all formal results on the detection of unsoundness introduced in this chapter.

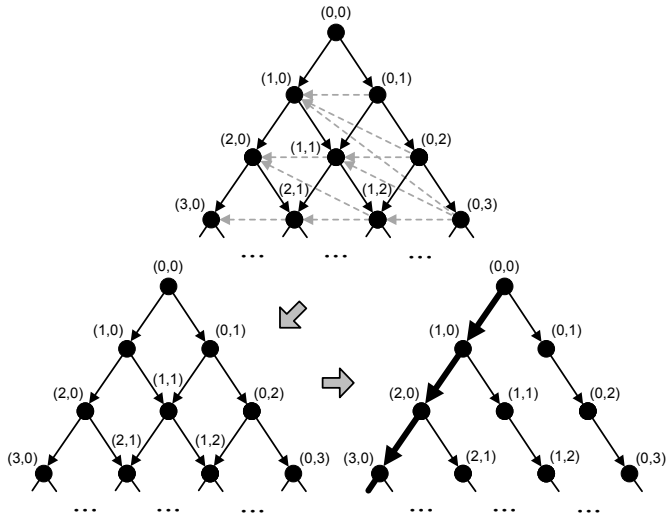
Despite the large body of related work on the formal verification of process models, we are not aware of any work that employs the connectivity property as an angle to their structural analysis in a systematic way. As shown in this chapter, stepwise decomposition based on connectivity is an effective and efficient way to address verification. Even though recent work showed impressive results in terms of soundness checking efficiency [41], the results have been achieved under optimizations for free-choice models. Our approach of stepwise decomposition, in turn, provides the foundations to tackle general classes of behavioral models. By employing the decomposition, we realize a *divide and conquer* verification strategy that can be combined with existing verification techniques to achieve a higher level of maturity in solutions to the problem of behavioral verification. For instance, our decomposition strategy can be combined with the works on reduction and inheritance rules to guide the application of these rules. While the decomposition is created using low-complexity algorithms, diagnostic information in case of unsoundness is provided as well.

A convenient property of a set of reduction rules is that of completeness. Completeness guarantees that a net can always be reduced with the help of the reduction rules to another smaller net which hints at certain interesting property of the net, e.g., liveness, boundedness, or soundness. The completeness of a set of reduction rules is a well-known problem, e.g., all of the above mentioned sets of rules are incomplete when applied to nets of an arbitrary structure. Moreover, given a set of rules, it is a challenge to decide in which order these rules must be applied to obtain the smallest reduced net, i.e., a net in which effects of the state

space explosion problem, when checking a property of interest, are decreased the most. We advocate the usage of a conceptually different approach. Instead of contributing to the interminable search for a complete set of *specific* reduction rules, we operate with a set of *generic* rules which are defined using the connectivity property of graphs. We always operate with complete sets of rules which have common structural characteristics (regardless of the net topology). Additionally, the connectivity property allows us to learn compositional structure of a net which suggests potential orders in which rules can be applied. Therefore, instead of searching for structural transformations which preserve an interesting property of a net, we check if this property can be deduced from a set of all subnets of the net; the subnets have common connectivity characteristics and collectively build up the net.

The results reported in this chapter have shown the usefulness of the connectivity property when checking soundness of WF-nets; the results range from the observation on cutvertices for the general class of nets to unique feedback on unsoundness for free-choice nets. We foresee the generalization of these results and introduction of new results for k-connected subnets as future steps.

8. Connectivity-Based Decomposition Framework



In this brief chapter, we reuse all the experience which we gained in the course of this thesis when working with the connectivity property of graphs and introduce the connectivity-based decomposition framework – a systematic approach for learning the compositional structure of behavioral models. This framework is developed as the generalization of the principles for the connectivity-based decomposition of behavioral models which we employed in Chapter 3 and Chapter 7. The connectivity-based decomposition framework defines a research agenda for developing new and improving existing methods which are founded on the structural decompositions of behavioral models.

The ideas presented in this chapter are published in [105, 115].

8.1. About this Chapter

Behavioral models, as addressed in this thesis, are annotated directed graphs with precise execution semantics. Graphs are mathematical objects with structural characteristics. Process analysts employ structural properties of graphs to formalize temporal orders of tasks in behavioral models. An interesting research aspect is the relation between the temporal relations of tasks in a behavioral model and the structural characteristics of the underlying graph.

In this thesis, we used the connectivity property of graphs when developing methods for parsing and analyzing behavioral models: In Chapter 3, we used the tree of the triconnected components of a behavioral model to discover all its canonical fragments. In Chapter 7, we used biconnected, triconnected, and 4-connected decompositions of WF-nets to derive conclusions about correctness, i.e., soundness, of the nets. The results for parsing behavioral models are employed when abstracting and structuring process models in Chapter 4 and Part III of this thesis. Furthermore, the results for the soundness verification of WF-nets can be reused when checking correctness of process models prior to their structuring.

The lesson learned from the above investigations is that connectivity-based decompositions can be employed to discover the compositional structure of behavioral models. Decompositions deliver information on separating elements of a model, its disconnected subgraphs, and structural relations between these subgraphs. This information can be used to define divide and conquer algorithms on behavioral models, as e.g., in Chapter 7. Moreover, individual decompositions can be composed into strategies (sequences of decomposition techniques), such that every discovered subgraph, due to some decomposition, gets decomposed again by the next technique from the sequence.

Different connectivity-based decomposition techniques stem from different configurations of elements, e.g., two vertices for the triconnected decomposition, that get removed in order to disconnect a (sub)graph. Every unique configuration results in a unique decomposition technique which can be employed to unveil unique structural characteristics of a graph. In this chapter, we utilize the lessons of previous chapters and propose a connectivity-based decomposition framework for behavioral models – a set of recommendations for composing decomposition strategies for behavioral models. The main idea of the framework is to decompose behavioral models in a way which allows the gradual discovery of subgraphs of higher connectivity. Please note that the framework does not provide precise instructions but rather gives suggestions on organizing decomposition strategies.

The next two sections revisit the connectivity property and the principles of the connectivity-based decomposition of graphs. The framework is described in Section 8.4. The chapter is wrapped up with the conclusion in Section 8.5.

8.2. Graph Connectivity Revisited

The classical concept of graph connectivity is based on the notion of graph elements, both vertices and edges. Recall from Section 3.2:

> An (undirected) (multi) graph is *k-connected* if there exists no set of $k-1$ elements, **each a vertex or an edge**, whose removal renders the graph disconnected. The *connectivity* of a graph is the largest k for which the graph is k-connected.

A 1-connected graph is often called *connected*. A graph is disconnected if there exists no path between some pair of elements of the graph. The notion of a path must ignore edge directions and can be seen as a sequence of alternating vertices and edges visited along a walk through the graph, where every two subsequent elements in the sequence are incident. Please also recall from Section 3.2 the following: Removal of a vertex from the graph implies removal of all the edges that are incident with the vertex. A graph composed of a single vertex is connected. A complete graph that is composed of n vertices, $n \geq 2$, is said to be $(n-1)$-connected.

So far in this thesis we employed the vertex connectivity – the connectivity property based on vertex removals only, see Chapter 3 and Chapter 7. One can certainly speak of "pure" vertex (or edge) connectivity.

> The *vertex (edge) connectivity* of a graph is the largest k for which there exist no set of $k-1$ vertices (edges) whose removal renders the graph disconnected.

An important result in graph theory is that for an arbitrary graph, it always holds that its vertex connectivity is less than or equal to its edge connectivity [23]. Intuitively, the above holds true, as the removal of an edge, when testing the edge connectivity, can be substituted by the removal of an incident vertex.

In order to pursuit for a more fine grained control of the connectivity property of graphs, we suggest to consider both vertices and edges, but insist on a clear distinction between vertex and edge removals. In the general case, one can speak about (v,e)-*connectedness* of a graph.

> An (undirected) (multi) graph is (v,e)-*connected* if there exists no set of v vertices and e edges whose removal renders the graph disconnected.

We exemplify all the above described concepts with the graph in Figure 8.1. The graph is 1-connected, not 2-connected and, hence, the connectivity of the graph is 1. The graph is connected if no elements are removed from the graph and becomes disconnected if vertex v_3 gets removed. The vertex connectivity of the graph, as explained above, is 1. At the same time, the edge connectivity of the graph is 2. The graph cannot be disconnected by the removal of a single edge, but requires the removal of at least two edges, e.g., $\{e_1, e_2\}$ or $\{e_4, e_5\}$. Finally, the graph in Figure 8.1 is clearly $(0,0)$-connected (it is connected if no elements are removed), but also $(0,1)$-connected.

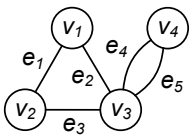


Figure 8.1. An undirected graph

In the next section, we shall employ the (v,e)-connectedness to revisit the connectivity-based decomposition principles of graphs.

8.3. Connectivity-Based Decomposition Revisited

The idea of the intended framework is to guide the decomposition of a behavioral model by stepwise varying the size of the set of elements that is used to disconnect the model (underlying graph). Prior to the introduction of the framework, in this section, we revisit the principles of the connectivity-based decomposition of graphs.

Trivially, one can perceive that a (v, e)-connected graph is not necessarily $(v + 1, e)$- or $(v, e + 1)$-connected. For instance, the $(0, 1)$-connected graph in Figure 8.1 is neither $(0, 2)$-connected, nor $(1, 1)$-connected; the elements that make the graph disconnected are, for instance, $\{e_1, e_2\}$ or $\{e_1, v_3\}$, respectively. Clearly, if the graph stays connected after the removal of any $v + e$ elements, it might get disconnected after the removal of some $v + e + 1$ elements. However, the graph in Figure 8.1 is also not $(1, 0)$-connected; vertex v_3 alone – when removed – disconnects the graph. Due to the fact that the vertex connectivity is always less than or equal to the edge connectivity of a graph [23] (see the discussion in the previous section), it holds that a (v, e)-connected graph is not necessarily $(v+1, f)$-connected, where $f \leq e$, or $(v, e+1)$-connected. Note that this rationale only makes sense under the assumption that – when iteratively testing the connectedness of a graph – one increases the number of either vertices or edges by exactly one element, which supports our intention to organize the stepwise decomposition of graphs.

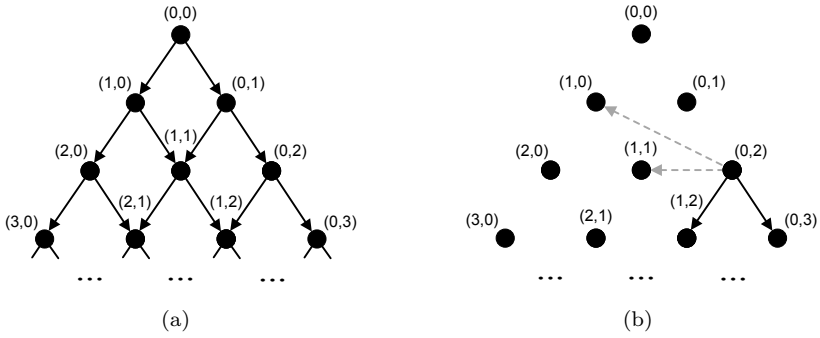


Figure 8.2. Guidelines for defining strategies when testing connectedness of graphs

A naïve approach for testing the (v, e)-connectedness of a graph can proceed by iteratively removing v vertices and e edges from the graph, for each such combination, and checking if the resulting graph is connected. Once it is known that a graph is (v, e)-connected, it can be challenged for being even "better" connected. A sequence of graph connectedness checks forms a *strategy*.

Figure 8.2 visualizes the principles for composing strategies for iteratively testing graph connectedness. In Figure 8.2(a) and Figure 8.2(b), every point represents a connectedness property. For instance, point $(0, 0)$ represents the $(0, 0)$-connected property of a graph. Every arc encodes the next property to be checked. For example, assume that we examine a graph which we know is $(0, 0)$-connected; this

corresponds to the topmost point $(0,0)$ in Figure 8.2(a). Knowing this, as the next step, we suggest to test the graph for being $(1,0)$- or $(0,1)$-connected (follow the arcs in Figure 8.2(a)). Therefore, Figure 8.2(a) can be interpreted as a guideline for defining strategies where (v,e)-connected graphs are suggested to be evaluated for being $(v+1,e)$- or $(v,e+1)$-connected. In addition to Figure 8.2(a), Figure 8.2(b) shows that a graph of a certain connectedness, in particular a $(0,2)$-connected graph, can be tested not only for being $(1,2)$- or $(0,3)$-connected, as already visualized in Figure 8.2(a), but also for being $(1,0)$- or $(1,1)$-connected, see the dashed arrows in the figure.

A *not* (v,e)-connected graph can be disconnected by removing v vertices and e edges, which gives a rise for a decomposition of the graph. The disconnected subgraphs of a graph – either with, or without the removed elements – are the products of a decomposition. A (v,e)-decomposition of a graph is a collection of graph subgraphs obtained by disconnecting the graph when removing v vertices and e edges; the subgraphs must be collected over all possible removals of v vertices and e edges. The notion of a strategy that we proposed for iterative tests of connectedness can as well be used to guide graph decompositions. A (v,e)-connected graph (represented by point (v,e) in Figure 8.2) can be (x,y)-decomposed, where (x,y) is the point in the figure that can be reached from point (v,e) by following on only one directed arc. Several decomposition steps form a *decomposition strategy*.

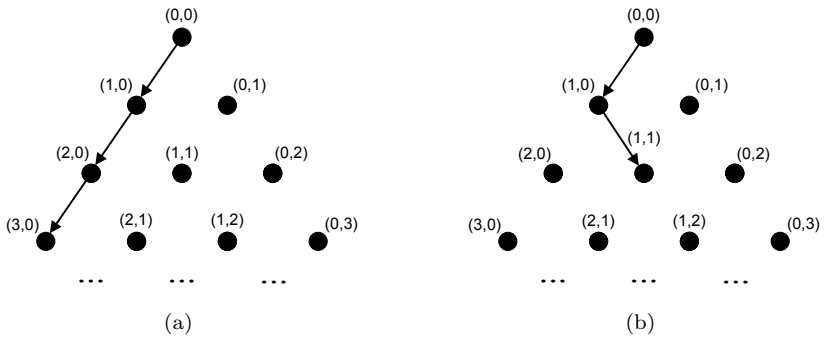


Figure 8.3. (a) Decomposition strategy for soundness verification from Chapter 7, and (b) the $(0,0)$,$(1,0)$,$(1,1)$ decomposition strategy

In Chapter 7, we employed the $(0,0)$,$(1,0)$,$(2,0)$,$(3,0)$ strategy to organize the soundness verification of WF-nets; the strategy is visualized in Figure 8.3(a). Point $(0,0)$ is a starting point (all WF-nets are connected). Points $(1,0)$, $(2,0)$, and $(3,0)$ correspond to decompositions employed at the biconnected, the triconnected, and the 4-connected steps, respectively, see Sections 7.4–7.6.

Figure 8.3(b) shows the decomposition strategy which proposes to start decomposing graphs by first removing a single vertex and afterwards to decompose derived subgraphs by removing one vertex and one edge. The strategy is exemplified in Figure 8.4 where we decompose the graph in Figure 8.1. Figure 8.4(a) shows the result of $(1,0)$-decomposition. The graph gets decomposed into two

subgraphs by removing vertex v_3. Figure 8.4(b), Figure 8.4(c), and Figure 8.4(d) show disconnected subgraphs obtained by removing one vertex and one edge from the subgraphs in Figure 8.4(a); the removed elements are $\{v_1, e_3\}$, $\{v_2, e_2\}$, and $\{v_3, e_1\}$, respectively. Please observe that in Figure 8.4(a) we preserved vertex v_3, whereas in other figures we completely removed the separating elements. A sequence of decisions along a decomposition strategy, on either to preserve or to remove the separating elements, can be seen as a property of the strategy.

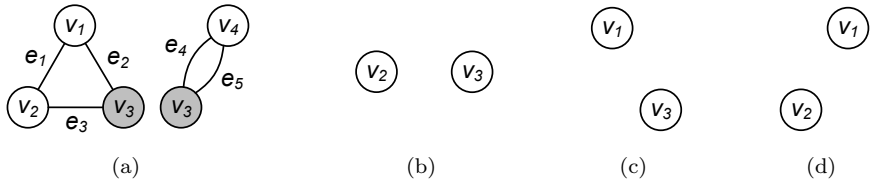


Figure 8.4. Decomposition strategy exemplified

The graph in Figure 8.1 can be decomposed by following different decomposition strategies. For example, after the $(0, 0)$ starting point, one can start by verifying that the graph is $(0, 1)$-connected and, afterwards, continue with either $(1, 1)$- or $(0, 2)$-decomposition. Every decomposition strategy results in different decomposition products, i.e., artifacts that unveil unique structural characteristics of the initial graph. We encourage the reader to try out different decomposition strategies as an exercise.

8.4. Decomposition Framework

After revisiting the notion of graph connectivity in Section 8.2 and the principles of the connectivity-based decomposition of graphs in Section 8.3, in this section we propose a framework to guide the connectivity-based decompositions of behavioral models. We believe that our framework can serve as a basis for the systematic structural analysis of behavioral models and can be of great help when developing new or rethinking existing methods which are founded on structural decompositions of behavioral models. Essentially, the framework accumulates the experience on the techniques for decomposing behavioral models that was gained in the course of this thesis, and is designed to promote certain decomposition strategies while suppressing the others.

Figure 8.5(a) shows the complete guidelines for testing connectedness/defining decomposition strategies of graphs, which follow the principles discussed in the previous section. The connectivity-based decomposition framework defines different priorities among all possible decomposition strategies, which are encoded by all possible paths in Figure 8.5(a). The framework is depicted in Figure 8.5(b). Just as before, points represent the connectedness property of graphs and arcs suggest which decompositions can be performed on graphs of a certain connectedness. For all decomposition strategies, we propose to give preference to those which are composed using vertex-based decompositions (refer to the left-most path in

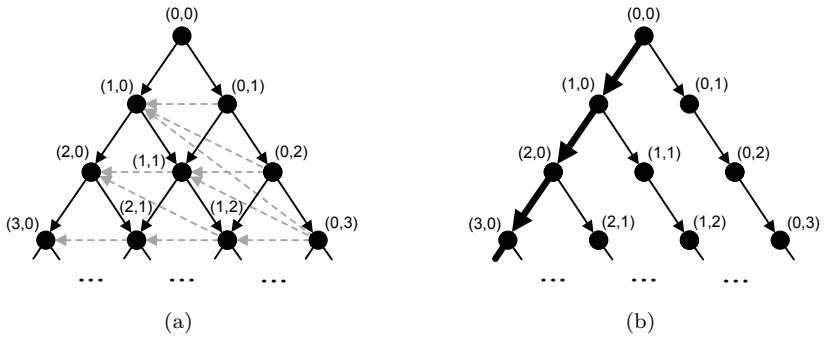


Figure 8.5. (a) Guidelines for defining decomposition strategies following on the principles described in Section 8.3, and (b) connectivity-based decomposition framework

Figure 8.5(b)). Such strategies stimulate the subgraphs to be discovered "faster", i.e., by performing less decomposition steps, and "finer", i.e., by discovering more subgraphs. This claim is supported by the experience gained when working on the algorithm for construction of RPST, where we used the vertex-based decomposition to discover *all* canonical fragments. However, in the situations when it is absolutely important to achieve granularity on the level of edges, we suggest deviating from the main strategy only once by switching from the vertex-based to the edge-based decomposition strategy. At the next level of priority we place those decompositions that are captured by the arcs shown in Figure 8.5(a), but not drawn in Figure 8.5(b). Finally, the lowest priority is assigned to decompositions that violate the principles of Section 8.3, e.g., a $(0,2)$-connected graph can be a subject for the $(2,0)$-decomposition.

So far, in this chapter, we assumed that all decompositions are carried out for undirected graphs. Behavioral models, like BPMN models, EPCs, WF-nets, or process models, are formalized as directed graphs. Edge directions are irrelevant for the purpose of the connectivity-based decompositions. However, edge directions become useful when classifying boundary nodes, i.e., nodes that reconnect disconnected subgraphs with the rest of the graph, as entries or exits. The classification can be accomplished using Definition 3.2.

Much of research was carried out by the compiler theory community to gain value from $(2,0)$- and $(0,2)$- decompositions of directed process graphs [124, 62] (for an elaborate discussion of these works please refer to Section 3.5). The subgraphs derived during $(2,0)$- and $(0,2)$- decompositions of directed graphs form hierarchies of SESE fragments. In our work, the $(2,0)$-decomposition plays a vital role in the algorithm for the construction of RPSTs, see the tree of the triconnected components in Section 3.2.3 and its role in Chapter 3.

A (v,e)-decomposition, where $v+e \geq 3$, allows one to decompose highly connected graphs into multi-entry-multi-exit (MEME) subgraphs, with up to $v+e$ entries and exits. For reasonable (v,e) combinations, i.e., when $v+e$ is sufficiently small, it is possible to perform decompositions in low polynomial-time complexity. For example, as explained in Section 7.6, the $(3,0)$-decomposition of a $(2,0)$-connected

graph can be accomplished by removing a vertex from the graph and afterwards running the $(2,0)$-decomposition, for which the linear time complexity algorithm exists [48]. Each discovered separation pair of vertices, together with the earlier removed vertex, form a separating triple of the initial graph. The procedure should be repeated for each vertex in the graph. Hence, a square-time complexity decomposition procedure can be obtained. Following the described rationale, one can accomplish $(v,0)$-decomposition of a graph in $O(n^{v-1})$ time, where n is proportional to the size of the graph.

8.5. Conclusion

In this chapter, we proposed a connectivity-based decomposition framework for behavioral models. As already mentioned in the preamble to this chapter, we understand the connectivity-based decomposition framework as the research agenda for methods which are founded on structural decompositions of behavioral models. The framework is composed of a set of recommendations on how to organize stepwise decompositions of behavioral models. It should be treated as guidelines for performing decompositions when improving existing or creating new methods that relate to the structure theory of behavioral models. The framework suggests giving the priority to the vertex-based decompositions over the edge-based ones. In those situations where one absolutely requires the granularity of the edge-based decompositions, the framework recommends composing strategies in a way that one never switches from the edge-based decompositions back to the vertex-based ones. The framework hints at the fact that decomposition strategies can also be composed of arbitrary decompositions which are arbitrarily ordered; however, such strategies are assigned the lowest priority by the framework.

The connectivity-based decomposition framework provides a convenient mechanism for categorizing techniques which are based on the structural decompositions of behavioral models. For example, the RPST algorithm in Chapter 3 is based on the $(1,0),(2,0)$ strategy, whereas the verification technique in Chapter 7 benefits from the $(0,0),(1,0),(2,0),(3,0)$ strategy. Furthermore, the framework suggests that, for instance, the verification technique in Chapter 7 can be extended by first looking at $(4,0)$- or $(3,1)$- decompositions of $(3,0)$ connected subgraphs, or one can decide to try with generalizing verification results for the family of $(v,0)$-decompositions. Finally, in order to achieve completeness when documenting decomposition strategies, individual decompositions can be annotated with additional properties, e.g., in the case of the soundness verification technique in Chapter 7 these properties are the descriptions on how subnets of a WF-net get transformed into sub-WF-nets prior to feeding them in for the next decomposition.

9. Conclusion

This chapter concludes the thesis: Section 9.1 summarizes the main contributions of the thesis; mainly, these are the techniques for structuring process models, but also those for parsing, abstraction, and verification of behavioral models. Afterwards, Section 9.2 lists open problems and research opportunities for proceeding with the research started in this thesis. Finally, Section 9.3 provides with references to implementations of the core results of the thesis.

9.1. Contributions of this Thesis

In this section, we once again name and give a brief summary of each of the main contributions of the thesis at hand.

Structuring Process Models

The core contribution of this thesis is a collection of techniques for structuring process models. Every single technique addresses the structuring of a specific class of process models (process components), and collectively these techniques cover the complete spectrum of the structuring problem. Individual structuring techniques address: structuring of acyclic process models, maximal-structuring of inherently unstructured acyclic process models, structuring of multi-source and/or multi-sink acyclic process models, and structuring of process models with cyclic paths.

Similar to other related works, this thesis confirms that not every process model can be structured and provides a characterization of the class of inherently unstructured process models. Therefore, given an unstructured process model as input and by using the results of this thesis, one is able to answer the question if the process model has an equivalent well-structured version, and, if the answer to this question is "yes" – one can construct the well-structured version of the input process model.

Proper Prefixes and Ordering Relations Graphs

The ideas presented in this thesis build upon and reuse techniques from several fields. The theories that we make the most use of are the graph theory, the theory of two-structures, the compiler theory, and the theory of distributed systems. One perspective on our results on structuring can be that they stem from a unique combination of standard techniques from different domains. Fundamentally, we showed how the semantics of distributed systems can be described in terms of graphs, or two-structures, and how one can benefit from this new representation. Along the path to this observation we defined the notion of the proper complete prefix unfolding. Proper complete prefix unfoldings are constructed by applying the new (restricted) criterion for truncating complete prefix unfoldings. This criterion is introduced in order to gain smooth round-tripping between the Petri net semantics and the partial order semantics of distributed systems. We also showed how the partial order semantics of a distributed system can be captured in a two-structure and consequently a graph. With this understanding in place, we were able to apply standard techniques to implement structuring of process models.

The notions of the proper complete prefix unfolding and the ordering relations graph are at the core of the proposed techniques for structuring process models. We believe that they can find their use in many other applications.

Parsing, Abstraction, and Verification of Behavioral Models

Along our research path towards the proposed structuring techniques, we worked on algorithms for parsing, abstraction, and verification of behavioral models. Parsing is a technique for discovering the compositional structure of behavioral models. This structural information can be of great use in various applications, but mainly to define *divide and conquer* algorithms aimed at the analysis of behavioral models. We used parsing to propose abstractions of process logic in behavioral models; parts of a behavioral model with clear interfaces get captured in subprocesses, which brings the original model to a higher abstraction level. Finally, we reused the experience on structural analysis of behavioral models gained when working on parsing and abstraction to propose the stepwise technique for verification of WF-nets. WF-nets get gradually decomposed by removing their nodes. Artifacts obtained along a decomposition of a WF-net are then used to judge the soundness of the net. All the above mentioned results are reported in the thesis.

We used parsing and abstraction of process models for modularization of the structuring problem; many secondary details can be abstracted away in process models so that one can concentrate on the essentials of structuring. The results of verification can be applied prior to structuring; indeed, process models that are provided for structuring must be sound. Despite how much we enjoy structuring, it is abundanly clear that the results on parsing, abstraction, and verification of behavioral models can be reused in many other contexts as well.

Connectivity-Based Decomposition Framework

In this thesis, we proposed the connectivity-based decomposition framework. The framework must be understood as a collection of high-level recommendations for proceeding with structural decompositions of behavioral models. The recommendations were accumulated in the course of the work on this thesis as lessons learned from employing structural decompositions for various applications in various contexts. The framework suggests which decompositions can be applied to a given behavioral model in order to observe more structural information about the model with the lowest computational effort. It is then left to a researcher to decide which decomposition strategy to choose for a model within the framework and how to benefit from obtained information; such decisions must be carried out based on the concrete problem at hand. The framework provides a convenient mechanism for documenting and thus comparing scientific results which are founded on structural decompositions of behavioral models.

9.2. Open Problems and Research Opportunities

In this work, not all aspects of the structuring problem were discussed at a sufficient level of detail. This section lists open problems and research opportunities which aim at improvements of the results proposed in this thesis, as well as states those opportunities that are inspired by this thesis. The target of the ideas is the generalization of applicability of structuring techniques, as well as their

optimization. We believe that advancements in the following areas can contribute considerably to the overall knowledge of the management of distributed systems.

Complex Gateway Types

Process models orchestrate tasks by means of *xor* and/or *and* gateways. One possibility for generalization of our structuring techniques comes from the introduction of complex gateway types (other than *xor* and *and*) in process models.

For instance, the proposed structuring techniques do not apply to process models with *or* gateways, except in the case where *or* gateways appear at the boundaries of a bond. We assume the semantics of *or* gateways from [136, 120], i.e., *or* splits correspond to Multi-Choice and *or* joins correspond to General Synchronizing Merge control flow patterns. According to [120], an *or* split describes a point in the process where a branch diverges into two or more branches such that when the incoming branch is enabled, the thread of control is immediately passed to one or more of the outgoing branches based on a mechanism that selects one or more outgoing branches, whereas an *or* join represents the convergence point of two or more branches which diverged earlier in the process into a single subsequent branch so that the thread of control is passed to the subsequent branch when either (i) each active incoming branch has been enabled or (ii) it is not possible that any branch that has not yet been enabled will be enabled at any future time.

When *or* gateways are present inside a rigid component, we believe that the structuring problem becomes conceptually similar to the problem of transforming a process model with *or* gateways into an FCB-equivalent process model with *xor* and *and* gateways only, so that the unfolding techniques can be applied.

For the comprehensive list of complex gateway types which can be used to generalize our notion of a process model please refer to [136, 120].

Canonical Process Models

A process can be seen as a partial order of tasks (transitions of a corresponding net). Process models describe processes as compositions of tasks by means of gateways and control flow arcs. An interesting theoretical problem to investigate is discovering the minimal set of tasks that can be composed in a process model which describes the same process as a given model. A process model with the minimal set of tasks can be seen as the *canonical version* of the given model. Tasks in canonical process models can be treated as having unique names (even if two tasks share the same name, e.g., two tasks with name a can be treated as being named a_1 and a_2, where a_1 and a_2 are semantically the same as a).

We envision that the canonical version of a given process model can be constructed by first unfolding and afterwards refolding the input process model, by following the principles similar to those described in Section 6.1 and Section 6.2. The unfolding step can be implemented as the computation of the complete prefix unfolding, which is finite for bounded systems (even for systems with cyclic paths). The folding step can be implemented similar as in Section 6.2.6 and Section 6.2.7; the modifications of the technique may stem from the necessity to preserve certain

properties in the folded nets. The properties to be preserved are the subject for future investigations.

It is worth pointing out that spurious control flow relations in unstructured process models may forbid their structuring. However, these relations can be suppressed in the canonical version of an unstructured process model prior to structuring. We envision that the canonical version of an unstructured process model can sometimes be well-structured.

Unfolding Free-Choice Systems

All the techniques for structuring process models proposed in this thesis, see Chapter 6, subsume the technique for the construction of proper prefixes for net systems. The running time to compute the complete prefix unfolding of a safe system, and hence to compute its proper prefix, has an upper bound of $O(|T| \cdot R^{\xi})$, where T is the set of transitions, R is the number of reachable markings, and ξ is the maximal size of the presets or postsets of the transitions in the originative system [38]. Note that the number of reachable markings of a net system is in the worst case exponential to the number of nodes and flow edges in the net. As all other structuring steps can be accomplished in polynomial or linear time to the size of the net, the construction of proper prefixes is the computational bottleneck of our structuring techniques.

Please recall that the semantics of process models is defined by means of a mapping to free-choice nets, see Section 2.5.2. For structuring, we require process models to be correct, i.e., sound and, hence, safe. Consequently, when structuring, we always operate with safe free-choice systems. The class of safe free-choice systems is the subclass of safe systems. Therefore, one can reasonably expect that the upper bound for computing proper prefixes of safe free-choice systems can be lower than $O(|T| \cdot R^{\xi})$. Proving this claim holds true would imply the discovery of a new and faster algorithm for computing proper prefixes and, thus, speeding up all our process model structuring techniques.

Cyclic Structuring

In Section 6.4, we sketched a technique for structuring process models with arbitrary cyclic paths. The technique relies on a conjecture that the proper prefix of a sound free-choice WF-system is finite, and a sketch to the proof for the fact that in the rewired proper prefixes concurrency is kept encapsulated.

Proper prefixes of acyclic net systems are clearly finite. However, in the case of cyclic net systems it must be shown that every run of a system terminates, either at conditions that correspond to sink places of the net or at a healthy cutoff event. Moreover, further work is required to gain a better understanding of formal properties of rewired proper prefixes. In our future work, we plan to conduct a formal investigation in support of our intuitive description of the technique for structuring cyclic process models.

We believe that the theory of net system unfoldings, and in particular the proper complete prefix unfoldings, can be of great use in many methods dealing with the management of distributed systems. In our opinion, the role of unfoldings is

still greatly underestimated by research communities that specialize on studies of distributed systems.

Soundness of Multi-Source and/or Multi-Sink Nets

In Section 6.3, we proposed a technique for structuring multi-source and multi-sink process models. As a prerequisite to structuring, we require process models to be correct, i.e., free from deadlocks and lack of synchronizations. In Section 6.3.3, we proposed to organize correctness checks of acyclic multi-source process models based on unfoldings of the augmented versions of the corresponding nets. An acyclic net with multiple source places and/or multiple sink places gets augmented to a net with a single source place so that the fresh nodes and arcs of the net encode all possible instantiations of the original net. One can then use the finite unfolding of the augmented net to conclude on the correctness of the originative process model.

An engineering challenge is to lift the results on process model correctness checks discussed above from unfoldings to (proper) complete prefix unfoldings. In order to accomplish this task, one must account for unfolding truncations at (healthy) cutoff events when adjusting the checks that were initially proposed for unfoldings. The solution to this challenge can result in improvements to our structuring techniques: (i) The correctness checks of multi-source and multi-sink process models and the construction of the proper complete prefix unfolding (required for structuring) can be performed simultaneously. (ii) One can perform soundness checks of cyclic nets based on finite (proper) complete prefix unfoldings and, thus, organize structuring of cyclic multi-source and multi-sink process models by combining ideas from Section 6.3 and Section 6.4.

On-Line Structuring Optimization

In the conclusion to Chapter 6, we briefly mentioned two heuristics for optimizing the structuring of process models. Both optimization techniques must be applied during the construction of the proper prefixes. In the following, we elaborate more on each of these techniques:

- The decision between continuing to work with an unstructured process model or to switch to its well-structured version is sometimes made based on the amount of task duplication in the structured process model. If the task duplication ratio is high, then it can be preferable to stay with the initial unstructured process model. The task duplication ratio is dictated by the amount of events in the proper prefix – those that correspond to observable transitions in the originative system. During construction, events are appended to the proper prefix iteratively. This construction principle leaves opportunity for optimization; the whole structuring algorithm can be halted if the duplication ratio of events that correspond to observable transitions gets larger than the predefined threshold.
- A rigid process component is inherently unstructured if and only if the modular decomposition of its orgraph contains a concurrent primitive, see

Section 6.1. As an optimization we foresee that the condition above can be verified on-line, i.e., during the construction of the proper prefix. More precisely, we believe that the fact that the orgraph contains a concurrent primitive can be verified based on the structure of the proper prefix each time a new event is appended to the branching process. We think that the existence of a concurrent primitive in the MDT of the orgraph can be checked by exploring the asymmetric choice property [61] of the branching processes along the construction path towards the proper prefix. The detailed investigation of this claim is left for future work. With such a technique in place, one can decide that a process model is inherently unstructured before the proper prefix is constructed and, thus, substantially decrease the overall structuring time.

Connectivity-Based Verification of Workflow Nets

In Chapter 7, we provided the foundations for the stepwise connectivity-based verification of WF-nets. We foresee many opportunities for generalizing the results proposed in Chapter 7. For instance, as a possibility for generalization of Theorem 7.5, we envision that the problem of soundness verification of a WF-net can be decomposed into several smaller problems: one to check the classical soundness of the root biconnected sub-WF-net, and one to check *up-to-k-soundness* of a sub-WF-net for every non-root biconnected sub-WF-net of the WF-net, where up-to-k-soundness should be defined similar to the way suggested in [4, 138]. The value of k must be determined individually for each sub-WF-net. Intuitively, the value of k for a sub-WF-net must be deduced from the *k-boundedness* of the cutvertex place of the WF-net that induces the sub-WF-net, where k-boundedness of a place is the maximal number of tokens that can reside at the place at all reachable markings of the WF-net. The progress along the ideas stated above can allow to relax the safeness requirement in Theorem 7.5.

In Chapter 7, we conducted soundness investigations based on k-connected subnets of WF-nets, where $k \leq 4$. The next research steps for connectivity-based verification of WF-nets will have to deal with obtaining results for highly connected subnets, i.e., $k > 4$, as well as searching for results that apply in the general case of k-connectivity.

Connectivity Theory of Behavioral Models

The connectivity-based decomposition framework, proposed in Chapter 8, can serve as a basis for the initiative on cataloging scientific results from various research fields which are founded on structural decompositions of behavioral models. Existing scientific results can be classified within the framework and re-thought for improvement or generalization along the recommendations suggested by the framework. Moreover, one can use the framework for deriving ideas when coming up with new techniques. To conclude, we foresee that the framework can become a central communication point for sharing knowledge on techniques for the structural analysis of behavioral models. We also envision that subsequent works on analysis techniques that follow the recommendations of the framework

can help with refining the framework, which in the end can produce a synergy effect.

Ordering Relations Graphs

As already mentioned in Section 9.1, in this thesis we benefited from the unique understanding of the semantics of distributed systems. This understanding rests on two notions: the notion of the proper complete prefix unfolding and the notion of the ordering relations graph. Orgraphs are compact representations of the behavioral information captured in proper prefixes, which allowed us to reuse standard techniques from graph theory to implement the structuring of process models. We envision that similar to our application of modular decomposition on orgraphs when structuring process models, one can try to look for value when applying other standard graph techniques on orgraphs; note that one can see orgraphs as two-structures and, hence, also apply techniques from the theory of two-structures when analyzing orgraphs. We believe that orgraphs are interesting artifacts by themselves and that future works on analysis of orgraphs can provide us with interesting insights into the management of distributed systems.

9.3. Implementation

The techniques proposed in this thesis are implemented in open source projects:

- `jbpt` (Business Process Technologies 4 Java). This project contains code developed for research purposes in the domain of business process management. The project is founded on a graph library specifically developed for a convenient reuse of algorithms. The project includes algorithms for parsing, abstraction, verification of behavioral models, and much more.

  `jbpt` is published under the GNU General Public License at:
  `http://code.google.com/p/jbpt`.

- `bpstruct` – A tool for structuring concurrent systems. `bpstruct` is a tool for transforming unstructured processes/programs/service compositions (models of concurrency) into well-structured ones. The transformations preserve concurrency in resulting well-structured models.

  `bpstruct` is published under the GNU General Public License at:
  `http://code.google.com/p/bpstruct`.

All the evaluations reported in this thesis were carried out by using the code published in the projects listed above.

Appendix – Proofs

A.1. Lemma 3.14: Relation between fragments of a TTG whose completed version is biconnected and fragments of its normalized version

Consider a single node-splitting step transforming a graph G into G', let x be the node that is split into nodes $*x$ and $x*$, and let e be the edge that is added between $*x$ and $x*$. We define the following mappings for the next lemma:

1. A mapping ψ maps a set F of edges of G' to a set $\psi(F)$ of edges of G by $\psi(F) = F \setminus \{e\}$.
2. A mapping ϕ maps a set of edges H of G to a set $\phi(H)$ of edges of G' by $\phi(H) = H \cup \{e\}$ if H has an incoming edge to x as well as an outgoing edge from x, and otherwise $\phi(H) = H$.

Now, we claim[1]:

Lemma 3.14 *Let ϕ and ψ be as defined above. We have:*

1. *If $F \neq \{e\}$ is a fragment of G', then $\psi(F)$ is a fragment of G.*
2. *If H is a fragment of G, then $\phi(H)$ is a fragment of G'.*
3. *If $F \neq \{e\}$ is a canonical fragment of G', then $\psi(F)$ is a canonical fragment of G.*
4. *If H is a canonical fragment of G, then there exists a canonical fragment F of G' such that $\psi(F) = H$.*

Proof. We prove each part separately.

1.,2. The proofs of these parts are derived by straightforward applications of the definitions.

3. Suppose $\psi(F)$ are not canonical. Then there exist a fragment H of G that overlaps with $\psi(F)$. We know from part 2 of this lemma that $\phi(H)$ is a fragment, and from $\psi(F) \subseteq F$ and $H \subseteq \phi(H)$ it follows that F and $\phi(H)$ overlap, which contradicts our assumption that F is canonical.

4. Let S_1 and S_2 be two fragments of G such that the exit v of S_1 is the entry of S_2. If $S = S_1 \cup S_2$ is a fragment, we say that S is a *sequence* and S_1 and S_2 are called *segments* of S. We also say that S_1 and S_2 are *in sequence*. It follows from the biconnectedness of $C(G)$ that the entry of S_1 and the

[1] The author wants to thank Hagen Völzer for his assistance with formalizing the proof of Lemma 3.14

exit of S_2 are then distinct. Furthermore, S is a fragment if and only if all nodes that are incident to v belong to S. A sequence S is *maximal* if S is not a segment of another sequence. We know that a sequence is a canonical fragment if and only if it is maximal [146].

We define the fragment F as follows. If H is a sequence – hence maximal – and x is a boundary node of H such that all outgoing edges or all incoming edges of x are inside H, then we set $F = H \cup \{e\}$. Otherwise, we set $F = \phi(H)$. To show that F is a canonical fragment, we consider both cases separately.

Let H be a maximal sequence and let x, without loss of generality, be an entry of H such that all outgoing edges of x are inside H. (The exit case is analogous.) We distinguish two cases.

1. An incoming edge of x is in H. Call this edge e_0. Then the sequence H can be divided into two segments S_1, S_2 such that $e_0 \in S_1$. Then $\phi(S_1)$ and $\phi(S_2)$ are both fragments. Moreover they are in sequence, that is, $F = \phi(S_1) \cup \phi(S_2)$ is a sequence. F must be a maximal sequence because otherwise H would not be a maximal sequence. Therefore, F is a canonical fragment.
2. No incoming edge of x is in H. As all outgoing edges of x are in H, H is in sequence with the trivial fragment $\{e\}$ in G'. Because H is a maximal sequence of G, $F = H \cup \{e\}$ is a maximal sequence of G'.

Now we consider the "otherwise" case, i.e., $F = \phi(H)$. We know from part 2 of this lemma that F is a fragment. Suppose that F were not canonical. Then, there is some fragment F' of G' such that F and F' overlap. Therefore, none of three sets $F \setminus F'$, $F \cap F'$ and $F' \setminus F$ are empty. Lets call an edge f *original* if $f \neq e$. If all three sets $F \setminus F'$, $F \cap F'$ and $F' \setminus F$ contain an original edge, then $H = \psi(F)$ and $\psi(F')$ also overlap, which contradicts H being canonical. Therefore, we have to prove that none of the three sets $F \setminus F'$, $F \cap F'$ and $F' \setminus F$ equals $\{e\}$. To derive a contradiction we suppose that this is the case. It follows immediately that x must then be a boundary node of H. We assume without loss of generality that x is an entry of H.

We know from previous results [146], that there are only two ways in which two fragments F, F' can overlap:

1. F and F' are two non-maximal sequences that share a common segment.
2. F and F' are separation components w.r.t. the same boundary pair $\{u, v\}$ that share a common separation class w.r.t. $\{u, v\}$. (F and F' are then special *bond fragments* in the terminology of [146].)

We consider these two cases now separately.

2. Consider the case $F \cap F' = \{e\}$. The boundary pair of $\{e\}$ is $\{*x, x*\}$, which is therefore also the common boundary pair of F and F'. It follows that x is a separation point of $C(G)$, contradicting our assumption that $C(G)$ is biconnected. The other two cases use exactly the same argument.

1. Let F and F' be two non-maximal sequences that share a common segment.
   a) If the shared segment is $\{e\}$, then $e \in F$ and because of the definition of ϕ, H contains an outgoing edge from x. Because $\{e\}$ and $F' \setminus \{e\}$ are two fragments in sequence, all the incident edges to $x*$ are in F', which contradicts that H contains an outgoing edge from x.
   b) Let $F \setminus F' = \{e\}$, then $e \in F$ and because of the definition of ϕ, H contains an incoming edge to x. That edge must be inside $F' \cap F$, the overlapping segment of the two sequences. It follows that $*x$ is a boundary node of this segment in G'. As $x*$ is a boundary of that segment, this contradicts the assumption that $C(G)$ is biconnected.
   c) Let $F' \setminus F = \{e\}$. As F does not contain e, we have $F = H$. As $F = H$ is a sequence by assumption, some outgoing edge of x is outside F (otherwise we would not be in the top-level "otherwise" case). Call this edge e_0. By assumption $x*$ is an interior node of F'. Then e_0 must be in $F' \setminus F$, which contradicts the initial assumption of this subcase. ▫

A.2. Theorem 5.14: Relation between fully concurrent bisimulation of two labeled occurrence systems and their λ-ordering relations

Theorem 5.14 *Let $S_1 = (N_1, M_1)$, $N_1 = (B_1, E_1, G_1, \mathcal{T}_1, \lambda_1)$, and $S_2 = (N_2, M_2)$, $N_2 = (B_2, E_2, G_2, \mathcal{T}_2, \lambda_2)$, be two labeled occurrence systems with natural markings and distinctive labelings. Let $E_1' \subseteq E_1$ and $E_2' \subseteq E_2$ be observable events of N_1 and N_2, respectively, such that there exists a bijection $\psi : E_1' \to E_2'$ for which holds $\lambda_1(e) = \lambda_2(\psi(e))$, for all $e \in E_1'$. Let $\mathcal{R}_{\lambda_1}$ and $\mathcal{R}_{\lambda_2}$ be the λ-ordering relations of N_1 and N_2, respectively. Then, it holds:*

$$S_1 \approx S_2 \Leftrightarrow \mathcal{R}_{\lambda_1} \cong \mathcal{R}_{\lambda_2}.$$

Proof. We prove each direction of the equality separately.

($\Rightarrow$) Let S_1 and S_2 be FCB-equivalent. We want to show that $\mathcal{R}_{\lambda_1} \cong \mathcal{R}_{\lambda_2}$. Let us assume that $S_1 \approx S_2$ holds, but $\mathcal{R}_{\lambda_1} \cong \mathcal{R}_{\lambda_2}$ does not hold. Furthermore, let us consider events $e_i^1, e_j^1 \in E_1'$ that are in one-to-one correspondence with events $e_i^2, e_j^2 \in E_2'$, i.e., $\psi(e_i^1) = e_i^2$ and $\psi(e_j^1) = e_j^2$. All scenarios can be reduced to the following two cases:

Case 1: ($e_i^1 \parallel_{N_1} e_j^1$ or $e_i^1 \rightsquigarrow_{N_1} e_j^1$, and $e_i^2 \#_{N_2} e_j^2$). If $e_i^1 \parallel_{N_1} e_j^1$ or $e_i^1 \rightsquigarrow_{N_1} e_j^1$, then there exists process π_1 of S_1 that contains e_i^1 and e_j^1. If $e_i^2 \#_{N_2} e_j^2$, then there exists no process π_2 of S_2 that contains e_i^2 and e_j^2.

Case 2: ($e_i^1 \rightsquigarrow_{N_1} e_j^1$, and $e_j^2 \rightsquigarrow_{N_2} e_i^2$ or $e_i^2 \parallel_{N_2} e_j^2$). Let π_1 be a process of S_1 that contains e_i^1 and e_j^1, and let π_2 be a process of S_2 that contains e_i^2 and e_j^2.

Then, there exists no $\phi \subseteq \psi$, such that ϕ is an order-isomorphism between λ-abstractions of π_1 and π_2.

In both cases we reach a contradiction, i.e., systems $S1$ and $S2$ cannot be FCB-equivalent if the λ-ordering relations are not isomorphic.

($\Leftarrow$) Let $\mathcal{R}_{\lambda_1} \cong \mathcal{R}_{\lambda_2}$. We want to show that S_1 and S_2 are FCB-equivalent. Let us assume that $\mathcal{R}_{\lambda_1} \cong \mathcal{R}_{\lambda_2}$ holds, but $S_1 \approx S_2$ does not hold. Then, for instance, there exists process π_1' of S_1, which has no corresponding order-isomorphic process of S_2. Suppose that π_1' has the minimal size among all such processes, i.e., any prefix of π_1' has a corresponding order-isomorphic process of S_2. Let π_1' be an extension of process π_1 of S_1 by exactly one observable event $e_j^1 \in E_1'$. Let π_2 be a process of S_2 that is order-isomorphic with π_1. Let $e_j^2 \in E_2'$ be in one-to-one correspondence with e_j^1, i.e., $\psi(e_j^1) = e_j^2$. All scenarios can be reduced to the following three cases:

Case 1: There exists process π_2' of S_2 that contains e_j^2 and is an extension of π_2 by one observable event. Moreover, there exists $e_i^1 \in E_1'$ in π_1, such that $e_i^1 \rightsquigarrow_{N_1} e_j^1$. However, it holds $e_i^2 \parallel_{N_2} e_j^2$, for $e_i^2 \in E_2'$, such that $\psi(e_i^1) = e_i^2$; otherwise there exists an order-isomorphism $\phi \subseteq \psi$ between π_1' and π_2'.

Case 2: There exists no process π_2' of S_2 that contains e_j^2 and is an extension of π_2. Moreover, there exists $e_i^1 \in E_1'$ in π_1, such that $e_i^1 \rightsquigarrow_{N_1} e_j^1$. However, it holds $e_i^2 \#_{N_2} e_j^2$, for $e_i^2 \in E_2'$, such that $\psi(e_i^1) = e_i^2$.

Case 3: There exists process π_2' of S_2 that contains e_j^2 and is an extension of π_2, but not by only one observable event. Then, there exists $e_k^2 \in E_2'$, such that $e_k^2 \rightsquigarrow_{N_2} e_j^2$ but π_2 does not contain e_k^2. However, $e_k^1 \in E_1'$, such that $\psi(e_k^1) = e_k^2$, is not in π_1' and, hence, $e_k^1 \not\rightsquigarrow_{N_1} e_j^1$.

In all three cases we reach a contradiction, i.e., the λ-ordering relations cannot be isomorphic if systems S_1 and S_2 are not FCB-equivalent. ▫

A.3. Theorem 6.14: Relation between the MDT of an orgraph and the RPST of a well-structured process model

Before we proceed with the proof of the theorem, we present an auxiliary proposition. The proposition summarizes relations between components of a process model and modules of an orgraph.

Proposition A.1: Let C_1 be a process component and let M_1 be the corresponding ordering relations graph. Let M_2 be an ordering relations graph and let C_2 be its corresponding process component.

1. If C_1 is trivial or polygon, then M_1 is linear.
2. If M_2 is linear, then there exists C_2 that is trivial or polygon.
3. If C_1 is *and* (*xor*) bond, then M_1 is *and* (*xor*) complete.
4. If M_2 is *and* (*xor*) complete, then there exists C_2 that is *and* (*xor*) bond. ⋆

Now, we are ready to continue with the theorem.

Theorem 6.14 *Let $\mathcal{G}$ be an ordering relations graph. The MDT of $\mathcal{G}$ contains no primitive module, iff there exists a well-structured process model PM such that $\mathcal{G}$ is the ordering relations graph of PM.*

Proof. Let $\mathcal{G} = (V, A, \mathcal{B}, \sigma)$ be an ordering relations graph.

($\Rightarrow$) Let $\mathcal{G}$ be such that the MDT of $\mathcal{G}$ contains no primitive module. We show now by structural induction on the MDT of $\mathcal{G}$ that there exists a well-structured process model PM such that $\mathcal{G}$ is the ordering relations graph of PM. The MDT of $\mathcal{G}$ can contain trivial, linear, or complete modules.

Base: If the MDT of $\mathcal{G}$ consists of a single module M, then M is a trivial module and PM is a process model composed of a single task.

Step: Let M be a module of the MDT of $\mathcal{G}$ such that every child module of M has a corresponding well-structured process component. If M is linear, then PM can be a trivial or polygon component composed from children of M, see (2) in Proposition A.1. If M is complete, then PM can be a bond process component, either *and* or *xor*, composed from process components that correspond to child modules of M, see (4) in Proposition A.1. In both cases, M has a corresponding well-structured process model (component).

Therefore, there exists a well-structured process model PM that is composed of process components which correspond to child modules of module V, such that $\mathcal{G}$ is the ordering relations graph of PM.

($\Leftarrow$) Let PM be a well-structured process model such that $\mathcal{G}$ is the ordering relations graph of PM. We show now by structural induction on the RPST of PM that the MDT of $\mathcal{G}$ has no primitive module. Because PM is well-structured, the RPST of PM has no rigid component.

Base: If PM is composed of a single task, then the corresponding ordering relations graph contains one trivial module.

Step: Let C be a process component of the RPST of PM such that every child process component of C has a corresponding ordering relations graph without a primitive module. If C is trivial or polygon, then $\mathcal{G}$ is either trivial or linear, see (1) in Proposition A.1. If C is bond, then $\mathcal{G}$ is complete, see (3) in Proposition A.1. In both cases, C has a corresponding ordering relations graph without a primitive module.

Therefore, the MDT of the ordering relations graph that corresponds to PM has no primitive module. ▫

Bibliography

[1] Web services business process execution language version 2.0. committee specification, January 2007.

[2] Business process model and notation (BPMN) version 2.0, January 2011.

[3] A. V. Aho and J. D. Ullman. *Foundations of Computer Science*. Computer Science Press, W. H. Freeman and Company, 1992.

[4] K. Barkaoui and L. Petrucci. Structural analysis of workflow nets with shared resources. In *Workshop on Workflow Management: Net-Based Concepts, Models, Techniques and Tools (WFM)*, pages 82–95, 1998.

[5] G. D. Battista and R. Tamassia. On-line graph algorithms with SPQR-trees. In *International Colloquium on Automata, Languages and Programming (ICALP)*, volume 443 of *Lecture Notes in Computer Science*, pages 598–611. Springer, 1990.

[6] G. D. Battista and R. Tamassia. On-line maintenance of triconnected components with SPQR-trees. *Algorithmica*, 15(4):302–318, 1996.

[7] R. Bergenthum, J. Desel, and S. Mauser. Comparison of different algorithms to synthesize a Petri net from a partial language. *Transactions on Petri Nets and Other Models of Concurrency (TOPNOC)*, 3:216–243, 2009.

[8] G. Berthelot. Checking properties of nets using transformation. In *Applications and Theory of Petri Nets (ICATPN/APN)*, volume 222 of *Lecture Notes in Computer Science*, pages 19–40. Springer, 1986.

[9] G. Berthelot. Transformations and decompositions of nets. In *Applications and Theory of Petri Nets (ICATPN/APN)*, volume 254 of *Lecture Notes in Computer Science*, pages 359–376. Springer, 1987.

[10] E. Best. Structure theory of Petri nets: the free choice hiatus. In *Advances in Petri Nets*, volume 254 of *Lecture Notes in Computer Science*, pages 168–205. Springer, 1987.

[11] E. Best, R. R. Devillers, A. Kiehn, and L. Pomello. Concurrent bisimulations in Petri nets. *Acta Informatica (ACTA)*, 28(3):231–264, 1991.

[12] E. Best and M. W. Shields. Some equivalence results for free choice nets and simple nets and on the periodicity of live free choice nets. In *Colloquium on Trees in Algebra and Programming (CAAP)*, volume 159 of *Lecture Notes in Computer Science*, pages 141–154. Springer, 1983.

[13] C. Böhm and G. Jacopini. Flow diagrams, turing machines and languages with only two formation rules. *Communications of the ACM (CACM)*, 9(5):366–371, 1966.

[14] J. A. Bondy and U. Murty. *Graph Theory*. Springer, 2008.

[15] G. Canfora, M. D. Penta, R. Esposito, and M. L. Villani. A framework for QoS-aware binding and re-binding of composite Web Services. *Journal of Systems and Software (JSS)*, 81(10):1754–1769, 2008.

[16] J. Cardoso, A. P. Sheth, J. A. Miller, J. Arnold, and K. Kochut. Quality of Service for workflows and Web Service processes. *Journal of Web Semantics (WS)*, 1(3):281–308, 2004.

[17] D. Cohn and R. Hull. Business artifacts: A data-centric approach to modeling business operations and processes. *IEEE Data Engineering Bulletin (DEBU)*, 32(3):3–9, 2009.

[18] J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev. Deriving Petri nets for finite transition systems. *IEEE Transactions on Computers (TC)*, 47(8):859–882, 1998.

[19] T. Curran, G. Keller, and A. Ladd. *SAP R/3 Business Blueprint: Understanding the Business Process Reference Model*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[20] F. de Montgolfier. *Décomposition Modulaire des Graphes*. PhD thesis, Université Montpellier, 2003.

[21] G. Decker and J. Mendling. Process instantiation. *Data & Knowledge Engineering (DKE)*, 68(9):777–792, 2009.

[22] J. Desel and J. Esparza. *Free Choice Petri Nets (Cambridge Tracts in Theoretical Computer Science)*. Cambridge University Press, 1995.

[23] R. Diestel. *Graph Theory*. Springer-Verlag, 2005.

[24] R. M. Dijkman, B. Gfeller, J. M. Küster, and H. Völzer. Identifying refactoring opportunities in process model repositories. *Information & Software Technology (INFSOF)*, 53(9):937–948, 2011.

[25] E. W. Dijkstra. Letters to the editor: Go To statement considered harmful. *Communications of the ACM (CACM)*, 11(3):147–148, 1968.

[26] M. Dumas, L. García-Bañuelos, A. Polyvyanyy, Y. Yang, and L. Zhang. Aggregate quality of service computation for composite services. In *International Conference on Service Oriented Computing (ICSOC)*, volume 6470 of *Lecture Notes in Computer Science*, pages 213–227, 2010.

[27] M. Dumas, W. M. P. van der Aalst, and A. H. M. ter Hofstede. *Process-Aware Information Systems: Bridging People and Software through Process Technology*. Wiley-Interscience, Hoboken, NJ, 2005.

[28] P. Effinger, M. Siebenhaller, and M. Kaufmann. An interactive layout tool for BPMN. In *Workshop on Advanced Issues of E-Commerce and Web/based Information Systems (CES)*, pages 399–406. IEEE Computer Society, 2009.

[29] A. Ehrenfeucht, T. Harju, and G. Rozenberg. Theory of 2-structures. In *International Colloquium on Automata, Languages and Programming (ICALP)*, volume 944 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 1995.

[30] A. Ehrenfeucht, T. Harju, and G. Rozenberg. *The Theory of 2-Structures: A Framework for Decomposition and Transformation of Graphs*. World Scientific, 1999.

[31] A. Ehrenfeucht and G. Rozenberg. Theory of 2-structures, Part I: Clans, basic subclasses, and morphisms. *Theoretical Computer Science (TCS)*, 70(3):277–303, 1990.

[32] A. Ehrenfeucht and G. Rozenberg. Theory of 2-structures, Part II: Representation through labeled tree families. *Theoretical Computer Science (TCS)*, 70(3):305–342, 1990.

[33] F. Elliger, A. Polyvyanyy, and M. Weske. On separation of concurrency and conflicts in acyclic process models. In *Enterprise Modelling and Information Systems Architectures (EMISA)*, volume 172 of *Lecture Notes in Informatics*, pages 25–36. GI, 2010.

[34] J. Engelfriet. Branching processes of Petri nets. *Acta Informatica (ACTA)*, 28(6):575–591, 1991.

[35] T. Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[36] J. Esparza and K. Heljanko. *Unfoldings – A Partial-Order Approach to Model Checking*. EATCS Monographs in Theoretical Computer Science. Springer, 2008.

[37] J. Esparza, S. Römer, and W. Vogler. An improvement of McMillan's unfolding algorithm. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 1055 of *Lecture Notes in Computer Science*, pages 87–106. Springer, 1996.

[38] J. Esparza, S. Römer, and W. Vogler. An improvement of McMillan's unfolding algorithm. *Formal Methods in System Design (FMSD)*, 20(3):285–310, 2002.

[39] J. Esparza and M. Silva. Circuits, handles, bridges and nets. In *Applications and Theory of Petri Nets (ATPN)*, volume 483 of *Lecture Notes in Computer Science*, pages 210–242. Springer, 1991.

[40] D. Fahland. *From Scenarios To Components*. PhD thesis, Humboldt-Universität zu Berlin and Technische Universiteit Eindhoven, 2010.

[41] D. Fahland, C. Favre, B. Jobstmann, J. Koehler, N. Lohmann, H. Völzer, and K. Wolf. Instantaneous soundness checking of industrial business process models. In *Business Process Management (BPM)*, volume 5701 of *Lecture Notes in Computer Science*, pages 278–293. Springer, 2009.

[42] D. Fahland, C. Favre, J. Koehler, N. Lohmann, H. Völzer, and K. Wolf. Analysis on demand: Instantaneous soundness checking of industrial business process models. *Data & Knowledge Engineering (DKE)*, 70(5):448–466, 2011.

[43] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[44] M. Fowler. *Analysis Patterns: Reusable Object Models*. Addison-Wesley, Menlo Park, CA, 1996.

[45] L. García-Bañuelos. Pattern identification and classification in the translation from BPMN to BPEL. In *OTM Conferences*, volume 5331 of *Lecture Notes in Computer Science*, pages 436–444. Springer, 2008.

[46] A. Gibbons. *Algorithmic Graph Theory*. Cambridge University Press, 1985.

[47] T. Gschwind, J. Koehler, and J. Wong. Applying patterns during business process modeling. In *Business Process Management (BPM)*, volume 5240 of *Lecture Notes in Computer Science*, pages 4–19. Springer, 2008.

[48] C. Gutwenger and P. Mutzel. A linear time implementation of SPQR-trees. In *Graph Drawing*, volume 1984 of *Lecture Notes in Computer Science*, pages 77–90. Springer, 2001.

[49] R. Hauser, M. Friess, J. M. Küster, and J. Vanhatalo. An incremental approach to the analysis and transformation of workflows using region trees. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (TSMC)*, 38(3):347–359, 2008.

[50] R. Hauser and J. Koehler. Compiling process graphs into executable code. In *Generative Programming and Component Engineering (GPCE)*, volume 3286 of *Lecture Notes in Computer Science*, pages 317–336. Springer, 2004.

[51] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM (CACM)*, 12(10):576–580, 1969.

[52] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[53] J. Hopcroft and R. E. Tarjan. Algorithm 447: Efficient algorithms for graph manipulation. *Communications of the ACM (CACM)*, 16(6):372–378, 1973.

[54] J. E. Hopcroft and R. E. Tarjan. Dividing a graph into triconnected components. *SIAM Journal on Computing (SIAMCOMP)*, 2(3):135–158, 1973.

[55] M. E. Hopkins. A case for the GOTO. In *ACM Annual Conference*, pages 787–790. ACM, 1972.

[56] J. J. Horning and B. Randell. Process structuring. *ACM Computing Surveys (CSUR)*, 5(1):5–30, 1973.

[57] S.-Y. Hwang, H. Wang, J. Srivastava, and R. A. Paul. A probabilistic QoS model and computation framework for Web Services-based workflows. In *International Conference on Conceptual Modeling (ER)*, volume 3288 of *Lecture Notes in Computer Science*, pages 596–609. Springer, 2004.

[58] S.-Y. Hwang, H. Wang, J. Tang, and J. Srivastava. A probabilistic approach to modeling and estimating the QoS of Web-Services-based workflows. *Information Sciences (ISCI)*, 177(23):5484–5503, 2007.

[59] M. C. Jaeger, G. Rojec-Goldmann, and G. Mühl. QoS aggregation for Web Service composition using workflow patterns. In *Enterprise Distributed Object Computing Conference (EDOC)*, pages 149–159. IEEE Computer Society, 2004.

[60] M. C. Jaeger, G. Rojec-Goldmann, and G. Mühl. QoS aggregation in Web Service compositions. In *e-Technology, e-Commerce and e-Service (EEE)*, pages 181–185. IEEE Computer Society, 2005.

[61] L. Jiao, T.-Y. Cheung, and W. Lu. On liveness and boundedness of asymmetric choice nets. *Theoretical Computer Science (TCS)*, 311(1–3):165–197, 2004.

[62] R. C. Johnson. *Efficient Program Analysis using Dependence Flow Graphs*. PhD thesis, Cornell University, Ithaca, NY, USA, 1995.

[63] R. C. Johnson, D. Pearson, and K. Pingali. The program structure tree: Computing control regions in linear time. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 171–185, 1994.

[64] G. Keller, M. Nüttgens, and A.-W. Scheer. Semantische Prozeßmodellierung auf der Grundlage 'Ereignisgesteuerter Prozeßketten (EPK)'. Veröffentlichungen des Instituts für Wirtschaftsinformatik (IWi), Universität des Saarlandes, January 1992. In German.

[65] G. Keller and T. Teufel. *SAP R/3 Process Oriented Implementation: Iterative Process Prototyping*. Addison-Wesley, 1998.

[66] P. Kemper. Linear time algorithm to find a minimal deadlock in a strongly connected free-choice net. In *Applications and Theory of Petri Nets (ATPN)*, volume 691 of *Lecture Notes in Computer Science*, pages 319–338. Springer, 1993.

[67] B. Kiepuszewski, A. H. M. ter Hofstede, and C. Bussler. On structured workflow modelling. In *Conference on Advanced Information Systems Engineering (CAiSE)*, volume 1789 of *Lecture Notes in Computer Science*, pages 431–445. Springer, 2000.

[68] B. Kiepuszewski, A. H. M. ter Hofstede, and W. M. P. van der Aalst. Fundamentals of control flow in workflows. *Acta Informatica (ACTA)*, 39(3):143–209, 2003.

[69] I. Kitzmann, C. König, D. Lübke, and L. Singer. A simple algorithm for automatic layout of BPMN processes. In *Workshop on Advanced Issues of E-Commerce and Web/based Information Systems (CES)*, pages 391–398. IEEE Computer Society, 2009.

[70] D. E. Knuth. *The Art of Computer Programming*, volume 1–4A. Addison-Wesley Professional, Boston, MA, USA, 1 edition, 2011.

[71] J. Koehler and R. Hauser. Untangling unstructured cyclic flows - a solution based on continuations. In *CoopIS/DOA/ODBASE*, volume 3290 of *Lecture Notes in Computer Science*, pages 121–138. Springer, 2004.

[72] D. Krafzig, K. Banke, and D. Slama. *Enterprise SOA: Service-Oriented Architecture Best Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[73] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Symposium on Principles of Programming Languages (POPL)*, pages 207–218, 1981.

[74] J. M. Küster, C. Gerth, A. Förster, and G. Engels. Detecting and resolving process model differences in the absence of a change log. In *Business Process Management (BPM)*, volume 5240 of *Lecture Notes in Computer Science*, pages 244–260. Springer, 2008.

[75] M. Laguna and J. Marklund. *Business Process Modeling, Simulation, and Design*. Prentice Hall, 2005.

[76] R. Laue and J. Mendling. The impact of structuredness on error probability of process models. In *Information Systems Technology and its Applications (UNISCON)*, volume 5 of *Lecture Notes in Business Information Processing*, pages 585–590. Springer, 2008.

[77] R. Liu and A. Kumar. An analysis and taxonomy of unstructured workflows. In *Business Process Management (BPM)*, volume 3649 of *Lecture Notes in Computer Science*, pages 268–284. Springer, 2005.

[78] N. Lohmann. Compliance by design for artifact-centric business processes. In *Business Process Management (BPM)*, volume 6896 of *Lecture Notes in Computer Science*, pages 99–115. Springer, 2011.

[79] N. Lohmann and J. Kleine. Fully-automatic translation of open workflow net models into simple abstract BPEL processes. In *Modellierung*, volume 127 of *LNI*, pages 57–72. GI, 2008.

[80] N. Lohmann and K. Wolf. Artifact-centric choreographies. In *International Conference on Service Oriented Computing (ICSOC)*, volume 6470 of *Lecture Notes in Computer Science*, pages 32–46. Springer, 2010.

[81] S. Mazanek and M. Hanus. Constructing a bidirectional transformation between BPMN and BPEL with a functional logic programming language. *Journal of Visual Languages and Computing (VLC)*, 22(1):66–89, 2011.

[82] R. M. McConnell and F. de Montgolfier. Linear-time modular decomposition of directed graphs. *Discrete Applied Mathematics (DAM)*, 145(2):198–209, 2005.

[83] K. L. McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In *Computer Aided Verification (CAV)*, volume 663 of *Lecture Notes in Computer Science*, pages 164–177. Springer, 1992.

[84] K. L. McMillan. A technique of state space search based on unfolding. *Formal Methods in System Design (FMSD)*, 6(1):45–65, 1995.

[85] S. Melzer and S. Römer. Deadlock checking using net unfoldings. In *Computer Aided Verification (CAV)*, volume 1254 of *Lecture Notes in Computer Science*, pages 352–363. Springer, 1997.

[86] J. Mendling, G. Neumann, and W. M. P. van der Aalst. Understanding the occurrence of errors in process models based on metrics. In *OTM Conferences*, volume 4803 of *Lecture Notes in Computer Science*, pages 113–130. Springer, 2007.

[87] K. Menger. Zur allgemeinen Kurventheorie. *Fund. Math.*, 10:96—115, 1927.

[88] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[89] R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[90] R. H. Möhring. Algorithmic aspects of comparability graphs and interval graphs. In *Graphs and Orders*, pages 41–101. D. Reidel, 1985.

[91] R. H. Möhring and F. J. Radermacher. Substitution decomposition for discrete structures and connections with combinatorial optimization. *Annals of Discrete*

Mathematics, 19:257–356, 1984.

[92] M. Montali, M. Pesic, W. M. P. van der Aalst, F. Chesani, P. Mello, and S. Storari. Declarative specification and verification of service choreographiess. *ACM Transactions on the Web (TWEB)*, 4(1), 2010.

[93] D. Moore, C. Musciano, M. J. Liebhaber, S. F. Lott, and L. Starr. ""GOTO considered harmful" considered harmful" considered harmful. *Communications of the ACM (CACM)*, 30(5):351—-355, 1987.

[94] D. Mukherjee, P. Jalote, and M. G. Nanda. Determining QoS of WS-BPEL compositions. In *International Conference on Service Oriented Computing (ICSOC)*, volume 5364 of *Lecture Notes in Computer Science*, pages 378–393. Springer, 2008.

[95] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[96] M. Nielsen, G. D. Plotkin, and G. Winskel. Petri nets, event structures and domains, Part I. *Theoretical Computer Science (TCS)*, 13:85–108, 1981.

[97] G. Oulsnam. Unravelling unstructured programs. *The Computer Journal (CJ)*, 25(3):379–387, 1982.

[98] C. Ouyang, M. Dumas, A. H. M. ter Hofstede, and W. M. P. van der Aalst. From BPMN process models to BPEL web services. In *International/European Conference on Web Services (ICWS)*, pages 285–292. IEEE Computer Society, 2006.

[99] C. Ouyang, M. Dumas, W. M. P. van der Aalst, A. H. M. ter Hofstede, and J. Mendling. From business process models to process-oriented software systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 19(1):2:1–2:37, 2009.

[100] D. A. Padua, D. J. Kuck, and D. H. Lawrie. High-speed multiprocessors and compilation techniques. *IEEE Trans. Computers*, 29(9):763–776, 1980.

[101] D. Park. *Concurrency and Automata on Finite Sequences*. Computer Science Department, University of Warwick, 1981.

[102] C. Peltz. Web services orchestration and choreography. *IEEE Computer (COMPUTER)*, 36(10):46–52, 2003.

[103] M. Pesic, H. Schonenberg, and W. M. P. van der Aalst. Declarative workflow. In *Modern Business Process Automation*, pages 175–201. Springer, 2010.

[104] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Institut für instrumentelle Mathematik, Bonn, Germany, 1962.

[105] A. Polyvyanyy. Structural abstraction of process specifications. In *ZEUS*, volume 563 of *CEUR Workshop Proceedings*, pages 73–79. CEUR-WS.org, 2010.

[106] A. Polyvyanyy, L. García-Bañuelos, and M. Dumas. Structuring acyclic process models. In *Business Process Management (BPM)*, volume 6336 of *Lecture Notes in Computer Science*, pages 276–293. Springer, 2010.

[107] A. Polyvyanyy, L. García-Bañuelos, and M. Dumas. Structuring acyclic process models. *Information Systems (IS)*, 2011. In Press, Accepted Manuscript.

[108] A. Polyvyanyy, L. García-Bañuelos, D. Fahland, and M. Weske. Maximal structuring of acyclic process models. *CoRR*, abs/1108.2384, 2011.

[109] A. Polyvyanyy, L. García-Bañuelos, and M. Weske. Unveiling hidden unstructured regions in process models. In *OTM Conferences*, volume 5870 of *Lecture Notes in Computer Science*, pages 340–356. Springer, 2009.

[110] A. Polyvyanyy, S. Smirnov, and M. Weske. Process model abstraction: A slider approach. In *Enterprise Distributed Object Computing Conference (EDOC)*, pages 325–331. IEEE Computer Society, 2008.

[111] A. Polyvyanyy, S. Smirnov, and M. Weske. The triconnected abstraction of process models. In *Business Process Management (BPM)*, volume 5701 of *Lecture Notes in Computer Science*, pages 229–244. Springer, 2009.

[112] A. Polyvyanyy, S. Smirnov, and M. Weske. *Handbook on Business Process Management, Volume 1*, chapter Business Process Model Abstraction, pages 149–166. Springer, 2010.

[113] A. Polyvyanyy, J. Vanhatalo, and H. Völzer. Simplified computation and generalization of the refined process structure tree. In *Web Services and Formal Methods (WS-FM)*, volume 6551 of *Lecture Notes in Computer Science*, pages 25–41. Springer, 2010.

[114] A. Polyvyanyy, M. Weidlich, and M. Weske. The biconnected verification of workflow nets. In *OTM Conferences*, volume 6426 of *Lecture Notes in Computer Science*, pages 410–418. Springer, 2010.

[115] A. Polyvyanyy, M. Weidlich, and M. Weske. Connectivity of workflow nets: The foundations of stepwise verification. *Acta Informatica (ACTA)*, 48(4):213–242, 2011.

[116] A. M. Rabinovich and B. A. Trakhtenbrot. Behaviour structures and nets. *Fundamenta Informaticae (FI)*, 11:357–404, 1988.

[117] M. Reichert and P. Dadam. $\text{ADEPT}_{\text{flex}}$-supporting dynamic changes of workflows without losing control. *Journal of Intelligent Information Systems (JIIS)*, 10(2):93–129, 1998.

[118] S. Rinderle, M. Reichert, and P. Dadam. Flexible support of team processes by adaptive workflow systems. *Distributed and Parallel Databases (DPD)*, 16(1):91–116, 2004.

[119] F. Rubin. "GOTO considered harmful" considered harmful. *Communications of the ACM (CACM)*, 30(3):195–196, 1987.

[120] N. Russell, A. H. M. ter Hofstede, W. M. P. van der Aalst, and N. Mulyar. Workflow control-flow patterns: A revised view. Technical Report BPM-06-22, BPMcenter.org, 2006. BPM Center Report.

[121] M. Siebenhaller and M. Kaufmann. Drawing activity diagrams. In *Software Visualization (SOFTVIS)*, pages 159–160. ACM, 2006.

[122] S. S. Skiena. *Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica*. Reading, MA: Addison-Wesley, 1990.

[123] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing (SIAMCOMP)*, 1(2):146–160, 1972.

[124] R. E. Tarjan and J. Valdes. Prime subprogram parsing of a program. In *Symposium on Principles of Programming Languages (POPL)*, pages 95–105, 1980.

[125] R. Uba, M. Dumas, L. García-Bañuelos, and M. L. Rosa. Clone detection in repositories of business process models. In *Business Process Management (BPM)*, Lecture Notes in Computer Science. Springer, 2011.

[126] S. Unger and F. Mueller. Handling irreducible loops: Optimized node splitting versus DJ-graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(4):299–333, 2002.

[127] J. Valdes. *Parsing Flowcharts and Series-Parallel Graphs*. PhD thesis, Stanford University, Stanford University, CA, USA, 1978.

[128] R. Valette. Analysis of Petri nets by stepwise refinements. *Journal of Computer and System Sciences (JCSS)*, 18(1):35–46, 1979.

[129] W. M. P. van der Aalst. Verification of workflow nets. In *Applications and Theory of Petri Nets (ICATPN)*, volume 1248 of *Lecture Notes in Computer Science*, pages 407–426. Springer, 1997.

[130] W. M. P. van der Aalst. The application of Petri nets to workflow management. *Journal of Circuits, Systems, and Computers (JCSC)*, 8(1):21–66, 1998.

[131] W. M. P. van der Aalst. Workflow verification: Finding control-flow errors using Petri-net-based techniques. In *Business Process Management (BPM)*, volume 1806 of *Lecture Notes in Computer Science*, pages 161–183. Springer, 2000.

[132] W. M. P. van der Aalst and T. Basten. Inheritance of workflows: An approach to tackling problems related to change. *Theoretical Computer Science (TCS)*, 270(1–2):125–203, 2002.

[133] W. M. P. van der Aalst and M. Pesic. DecSerFlow: Towards a truly declarative service flow language. In *The Role of Business Processes in Service Oriented Architectures*, volume 06291 of *Dagstuhl Seminar Proceedings*, 2006.

[134] W. M. P. van der Aalst, M. Pesic, and H. Schonenberg. Declarative workflows: Balancing between flexibility and support. *Computer Science - Research and Development (IFE)*, 23(2):99–113, 2009.

[135] W. M. P. van der Aalst and C. Stahl. *Modeling Business Processes: A Petri Net-Oriented Approach*. Cambridge MA: MIT Press, 2011.

[136] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distributed Parallel Databases*, 14(1):5–51, 2003.

[137] W. M. P. van der Aalst, B. F. van Dongen, J. Herbst, L. Maruster, G. Schimm, and A. J. M. M. Weijters. Workflow mining: A survey of issues and approaches. *Data & Knowledge Engineering (DKE)*, 47(2):237–267, 2003.

[138] W. M. P. van der Aalst, K. M. van Hee, A. H. M. ter Hofstede, N. Sidorova, H. M. W. Verbeek, M. Voorhoeve, and M. T. Wynn. Soundness of workflow nets: Classification, decidability, and analysis. *Formal Aspects of Computing (FAC)*, 23(3):333–363, 2011.

[139] R. J. van Glabbeek. The linear time-branching time spectrum (extended abstract). In *International Conference on Concurrency Theory (CONCUR)*, volume 458 of *Lecture Notes in Computer Science*, pages 278–297. Springer, 1990.

[140] R. J. van Glabbeek. What is branching time semantics and why to use it? *Bulletin of the EATCS*, 53:191–198, 1994.

[141] R. J. van Glabbeek and U. Goltz. Equivalence notions for concurrent systems and refinement of actions (extended abstract). In *Mathematical Foundations of Computer Science (MFCS)*, volume 379 of *Lecture Notes in Computer Science*, pages 237–248. Springer, 1989.

[142] R. J. van Glabbeek and F. W. Vaandrager. Petri net models for algebraic theories of concurrency. In *Parallel Architectures and Languages Europe (PARLE)*, volume 259 of *Lecture Notes in Computer Science*, pages 224–242. Springer, 1987.

[143] K. M. van Hee, N. Sidorova, and M. Voorhoeve. Soundness and separability of workflow nets in the stepwise refinement approach. In *Applications and Theory of Petri Nets (ICATPN/APN)*, volume 2679 of *Lecture Notes in Computer Science*, pages 337–356. Springer, 2003.

[144] J. Vanhatalo. *Process Structure Trees: Decomposing a Business Process Model into a Hierarchy of Single-Entry-Single-Exit Fragments*. PhD thesis, University of Stuttgart, Germany, July 2009. Volume 1573, dissertation.de — Verlag im Internet. ISBN: 978-3-86624-473-3.

[145] J. Vanhatalo, H. Völzer, and J. Koehler. The refined process structure tree. In *Business Process Management (BPM)*, volume 5240 of *Lecture Notes in Computer Science*, pages 100–115. Springer, 2008.

[146] J. Vanhatalo, H. Völzer, and J. Koehler. The refined process structure tree. *Data & Knowledge Engineering (DKE)*, 68(9):793–818, 2009.

[147] J. Vanhatalo, H. Völzer, and F. Leymann. Faster and more focused control-flow analysis for business process models through SESE decomposition. In *International Conference on Service Oriented Computing (ICSOC)*, volume 4749 of *Lecture Notes in Computer Science*, pages 43–55. Springer, 2007.

[148] J. Vanhatalo, H. Völzer, F. Leymann, and S. Moser. Automatic workflow graph refactoring and completion. In *International Conference on Service Oriented Computing (ICSOC)*, volume 5364 of *Lecture Notes in Computer Science*, pages 100–115, 2008.

[149] E. Verbeek, M. T. Wynn, W. M. P. van der Aalst, and A. H. M. ter Hofstede. Reduction rules for reset/inhibitor nets. *Journal of Computer and System Sciences (JCSS)*, 76(2):125–143, 2010.

[150] H. M. W. E. Verbeek, T. Basten, and W. M. P. van der Aalst. Diagnosing workflow processes using woflan. *The Computer Journal (CJ)*, 44(4):246–279, 2001.

[151] H. M. W. E. Verbeek and W. M. P. van der Aalst. Woflan 2.0: A Petri-net-based workflow diagnosis tool. In *Applications and Theory of Petri Nets (ICATPN/APN)*, pages 475–484, 2000.

[152] B. Weber, M. Reichert, J. Mendling, and H. A. Reijers. Refactoring large process model repositories. *Computers in Industry*, 62(5):467–486, 2011.

[153] M. Weidlich. *Behavioural Profiles: A Relational Approach to Behaviour Consistency*. PhD thesis, University of Potsdam, 2011.

[154] M. Weidlich, G. Decker, A. Großkopf, and M. Weske. BPEL to BPMN: The myth of a straight-forward mapping. In *OTM Conferences*, volume 5331 of *Lecture Notes in Computer Science*, pages 265–282. Springer, 2008.

[155] M. Weske. *Business Process Management: Concepts, Languages, Architectures*. Springer Verlag, 2007.

[156] M. H. Williams. Generating structured flow diagrams: The nature of unstructuredness. *The Computer Journal (CJ)*, 20(1):45–50, 1977.

[157] G. Winskel. A new definition of morphism on Petri nets. In *Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 166 of *Lecture Notes in Computer Science*, pages 140–150. Springer, 1984.

[158] K. Wolf. Generating Petri net state spaces. In *Applications and Theory of Petri Nets (ICATPN/APN)*, volume 4546 of *Lecture Notes in Computer Science*, pages 29–42. Springer, 2007.

[159] W. A. Wulf. A case against the GOTO. In *ACM Annual Conference*, pages 791–797. ACM, 1972.

[160] M. T. Wynn, E. Verbeek, W. M. P. van der Aalst, A. H. M. ter Hofstede, and D. Edmond. Reduction rules for YAWL workflows with cancellation regions and OR-joins. *Information & Software Technology (INFSOF)*, 51(6):1010–1020, 2009.

[161] M. T. Wynn, E. Verbeek, W. M. P. van der Aalst, A. H. M. ter Hofstede, and D. Edmond. Soundness-preserving reduction rules for reset workflow nets. *Information Sciences (ISCI)*, 179(6):769–790, 2009.

[162] F. Zhang and E. H. D'Hollander. Using hammock graphs to structure programs. *IEEE Transactions on Software Engineering (TSE)*, 30(4):231–245, 2004.

Acknowledgements

When I look back into my life, I find it difficult to answer the question of how it happened that I am writing these lines now. The simple way is to say that I first went to school, then to university, and finally here I go with this thesis. Ah yes, there was a kindergarten in the very beginning! However, I cannot but admit the feeling of guidance; there must be a reason why all those random variables were drawn like that – at least, I like the feeling when thinking this way. It is true that I did my best to influence the process, but it is false that I had full control over what happened to me.

My interest in informatics started at about the age of six when the family of my friends brought a personal computer from a trip to East Germany. We used it to play games by loading them from audio tapes. Prior to getting a game running, we had to listen weird sounds for several minutes. Later in school, I started to program in BASIC and Pascal; in my senior class – Java and C++.

I completed my bachelor of science in computer science at NaUKMA, Kyiv. I want to use this chance to acknowledge my years in Kyiv and to thank the faculty of informatics at NaUKMA for the excellent classical education that was pushed on me. My special thanks are addressed to Volodymyr Boublik, who helped me with setting up priorities in my professional development. I continued with my postgraduate education at the HPI, Potsdam, where I completed my master in software engineering. It is at the HPI where my interest in different models of concurrency developed to the point when I realized that I would like to continue working on a thesis in this area.

When I look back at the last four years of my research, I have to admit that achieved results would not be in range without people around me. I am thankful to Mathias Weske, my supervisor, for hosting me at the business process technology group, for providing me with freedom when choosing the topic of research, and for his support during all my time with the group. I enjoyed these four years! Thank you! I am grateful to Marlon Dumas and Luciano García-Bañuelos who brought my attention back to the problem of structuring process models so that it never got blurred again. Constant feedback from Marlon and Luciano allowed me to see problems differently and to find solutions faster. I enjoyed the work with Hagen Völzer and Jussi Vanhatalo on parsing workflow graphs. This work helped me with setting up the stage for the structuring problem. Moreover, I learned from Hagen a lot on how a high-standard research should be conducted. Finally, I would like to thank Dirk Fahland for his work on refolding the unfoldings; it provided me with the last piece in the puzzle under the name the technique for maximal-structuring of acyclic process models.

I owe much to the members of the business process technology group. We started together in the group with Sergey by looking at the topic of business process model abstraction. Matthias shared his ideas on behavioral profiles with me; in some respect this collaboration led me to the definition of a notion of the ordering relations graph. Much feedback to my work was collected at the meetings that regularly took place in the group. For this input I am thankful to Ahmed, Evellin,

Gero, Rami, Nico, Matthias, Dominik, Alexander, Andreas, Harald, Adela, Hagen, Emilian, Frank, Andreas, Sergey, and Matthias.

Most important, I am thankful to my family, to my wife Anna for her love and support, to my parents Serhiy and Viktoria because without them I would not be me, and to my little sister Anna who will always stay little to me. Without the love of my family the world would be empty. Thank you for accepting me the way I am!

When I look forward into my life, I am excited about the new challenges it will pose. Every challenge will be conquered and every challenge will provide me with a valuable experience – at least, I like the feeling when thinking this way.

Publications

The results reported in the thesis at hand evolved from, or can be directly found in, the following publications.

Book Chapters

- Artem Polyvyanyy and Mathias Weske. *Flexible Service Systems*. In: The Science of Service Systems, Springer, 2011
- Artem Polyvyanyy, Sergey Smirnov, and Mathias Weske. *Business Process Model Abstraction*. In: International Handbook on Business Process Management, Springer, 2010

Journal Articles

- Artem Polyvyanyy, Luciano García-Bañuelos, and Marlon Dumas. *Structuring Acyclic Process Models*. Information Systems (IS). In Press, Accepted Manuscript
- Artem Polyvyanyy, Matthias Weidlich, and Mathias Weske. *Connectivity of Workflow Nets: The Foundations of Stepwise Verification*. Acta Informatica (ACTA), 48(4), 2011
- Matthias Weidlich, Artem Polyvyanyy, Jan Mendling, and Mathias Weske. *Causal Behavioural Profiles – Efficient Computation, Applications, and Evaluation*. Fundamenta Informaticae (FI). In Press, Accepted Manuscript
- Matthias Weidlich, Artem Polyvyanyy, Nirmit Desai, Jan Mendling, and Mathias Weske. *Process Compliance Analysis based on Behavioural Profiles*. Information Systems (IS), 36(7), 2011

Conference Papers

- Dieter Schuller, Artem Polyvyanyy, Luciano García-Bañuelos, and Stefan Schulte. *Optimization of Complex QoS-aware Service Compositions*. Proceedings of the 9th International Conference on Service Oriented Computing (ICSOC). Paphos, Cyprus, December 2011
- Marlon Dumas, Luciano García-Bañuelos, Artem Polyvyanyy, Yong Yang, and Liang Zhang. *Aggregate Quality of Service Computation for Composite Services*. Proceedings of the 8th International Conference on Service Oriented Computing (ICSOC). San Francisco, CA, US, December 2010
- Artem Polyvyanyy, Matthias Weidlich, and Mathias Weske. *The Biconnected Verification of Workflow Nets*. Proceedings of the 18th International Conference on Cooperative Information Systems (CoopIS). Crete, Greece, October 2010
- Artem Polyvyanyy, Luciano García-Bañuelos, and Marlon Dumas. *Structuring Acyclic Process Models*. Proceedings of the 8th International Conference on Business Process Management (BPM). Hoboken, NJ, US, September 2010 (*best paper award*)
- Matthias Weidlich, Artem Polyvyanyy, Jan Mendling, and Mathias Weske. *Efficient Computation of Causal Behavioural Profiles using Structural Decomposition*. Proceedings of the 31st International Conference on Application and Theory of Petri Nets and Other Models of Concurrency (ATPN). Braga, Portugal, June 2010
- Matthias Weidlich, Artem Polyvyanyy, Nirmit Desai, and Jan Mendling. *Process Compliance Measurement based on Behavioural Profiles*. Proceedings of the 22nd

International Conference on Advanced Information Systems Engineering (CAiSE). Hammamet, Tunisia, June 2010

- Artem Polyvyanyy, Luciano García-Bañuelos, and Mathias Weske. *Unveiling Hidden Unstructured Regions in Process Models*. Proceedings of the 17th International Conference on Cooperative Information Systems (CoopIS). Vilamoura, Algarve, Portugal, November 2009
- Artem Polyvyanyy, Sergey Smirnov, and Mathias Weske. *The Triconnected Abstraction of Process Models*. Proceedings of the 7th International Conference on Business Process Management (BPM). Ulm, Germany, September 2009
- Artem Polyvyanyy, Sergey Smirnov, and Mathias Weske. *On Application of Structural Decomposition for Process Model Abstraction*. Proceedings of the 2nd International Conference on Business Process and Services Computing (BPSC). Leipzig, Germany, March 2009
- Artem Polyvyanyy and Mathias Weske. *Flexible Process Graph: A Prologue*. Proceedings of the 16th International Conference on Cooperative Information Systems (CoopIS). Monterrey, Mexico, November 2008
- Artem Polyvyanyy, Sergey Smirnov, and Mathias Weske. *Process Model Abstraction: A Slider Approach*. Proceedings of the 12th International Conference on Enterprise Distributed Object Computing (EDOC). München, Germany, September 2008
- Ahmed Awad, Artem Polyvyanyy, and Mathias Weske. *Semantic Querying of Business Process Models*. Proceedings of the 12th International Conference on Enterprise Distributed Object Computing (EDOC). München, Germany, September 2008
- Ralf Knackstedt, Dominik Kuropka, Oliver Müller, and Artem Polyvyanyy. *An Ontology-Based Service Discovery Approach for the Provisioning of Product-Service Bundles*. In 16th European Conference on Information Systems (ECIS). Galway, Ireland, June 2008

Workshop Papers

- Artem Polyvyanyy, Jussi Vanhatalo, and Hagen Völzer. *Simplified Computation and Generalization of the Refined Process Structure Tree*. Proceedings of the 7th International Workshop on Web Services and Formal Methods (WS-FM). Hoboken, NJ, US, September 2010
- Marlon Dumas, Luciano García-Bañuelos, and Artem Polyvyanyy. *Unraveling Unstructured Process Models*. Proceedings of the 2nd International Workshop on BPMN (BPMN). Potsdam, Germany, October 2010 (*invited paper*)
- Felix Elliger, Artem Polyvyanyy, and Mathias Weske. *On Separation of Concurrency and Conflicts in Acyclic Process Models*. Proceedings of the 4th International Workshop on Enterprise Modelling and Information Systems Architectures. Karlsruhe, Germany, October, 2010
- Artem Polyvyanyy. *Structural Abstraction of Process Specifications*. 2nd Central-European Workshop on Services and their Composition (ZEUS). Berlin, Germany, February 2010 (*best presentation award*)
- Artem Polyvyanyy, Sergey Smirnov, and Mathias Weske. *Reducing Complexity of Large EPCs*. Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten (MobIS: EPK). Saarbrücken, Germany, November 2008
- Artem Polyvyanyy and Mathias Weske. *Hypergraph-Based Modeling of Ad-Hoc Business Processes*. Proceedings of the 1st International Workshop on Process

Management for Highly Dynamic and Pervasive Scenarios (BPM: PM4HDPS). Milan, Italy, September 2008

Technical Reports

- Artem Polyvyanyy, Luciano García-Bañuelos, Dirk Fahland, and Mathias Weske. *Maximal Structuring of Acyclic Process Models.* The Computing Research Repository (CoRR). August 2011
- Artem Polyvyanyy. *Parsing Behavior: The Hierarchical Nature of Concurrent Systems.* Proceedings of the 5th Ph.D. Retreat of the HPI Research School on Service-oriented Systems Engineering. October 2010
- Matthias Weidlich, Artem Polyvyanyy, Jan Mendling, and Mathias Weske. *Efficient Computation of Causal Behavioural Profiles using Structural Decomposition.* Technical Report of the Business Process Technology group, 10, 2010
- Artem Polyvyanyy, Jussi Vanhatalo, and Hagen Völzer. *Simplified Computation and Generalization of the Refined Process Structure Tree.* IBM Research Report, RZ3745, September 2009
- Artem Polyvyanyy and Dominik Kuropka. *A Quantitative Evaluation of the Enhanced Topic-Based Vector Space Model.* Technical Report of the Hasso Plattner Institute, 19, 2007

Curriculum Vitæ – Artem Polyvyanyy

9 Jul 1983	born in Mariupol, Ukraine
Sep 1993 – May 1999	student at high school "Humanitarian Gymnasium" with advanced language courses, Rivne, Ukraine
Sep 1999 – May 2000	student at high school "Basking-Ridge High School", Basking-Ridge County, New Jersey, USA
Sep 1999 – May 2000	grant of Freedom Support Act, FLEX-ACCELS international students exchange program
Sep 2000 – Jul 2004	student of Computer Science at the "National University of Kyiv-Mohyla Academy", Kyiv, Ukraine; specialization in Information Control Systems and Technologies; Bachelor of Science degree
Sep 2001 – May 2003	student assistant at the "Information Computer Center" (ICC), "National University of Kyiv-Mohyla Academy", Kyiv, Ukraine
Jun 2004 – Nov 2004	internship at "Wincor-Nixdorf International GmbH", software quality department, retail division, Hamburg, Germany
Apr 2005 – Apr 2007	student of IT-Systems Engineering at the "Hasso Plattner Institute" for Software Systems Engineering at the University of Potsdam, Germany; Master of Science degree
Apr 2005 – Apr 2007	scholarship of the German Academic Exchange Service – Deutscher Akademischer Austausch Dienst (DAAD)
Sep 2005 – Feb 2006	student assistant at the "Business Process Technology" group (Prof. Dr. Mathias Weske), "Hasso Plattner Institute" for Software Systems Engineering at the University of Potsdam, Germany
Mar 2007 – Sep 2007	internship at "SAP Labs", imagineering department, Palo Alto, USA
since Oct 2007	research assistant and a Ph.D. candidate in the "Business Process Technology" group (Prof. Dr. Mathias Weske), "Hasso Plattner Institute" for Software Systems Engineering at the University of Potsdam, Germany
since Oct 2007	member of the research school on "Service-Oriented Systems Engineering" at the "Hasso Plattner Institute" for Software Systems Engineering at the University of Potsdam, Germany

Tabellarischer Lebenslauf – Artem Polyvyanyy

9 Jul 1983	geboren in Mariupol, Ukraine
Sep 1993 – Mai 1999	Besuch des "Humanitarian Gymnasium" mit erweitertem Sprachunterricht, Rivne, Ukraine
Sep 1999 – Mai 2000	Besuch der "Basking-Ridge High School", Basking-Ridge County, New Jersey, USA
Sep 1999 – Mai 2000	Förderung im Rahmen des Freedom Support Act, FLEX-ACCELS, Internationales Schüleraustauschprogramm
Sep 2000 – Jul 2004	Studium der Informatik an der "National University of Kyiv-Mohyla Academy", Kyiv, Ukraine; Studienschwerpunkt in "Information Control Systems and Technologies"; Abschluss als Bachelor of Science
Sep 2001 – Mai 2003	Studentische Hilfskraft am "Information Computer Center" (ICC), "National University of Kyiv-Mohyla Academy", Kyiv, Ukraine
Jun 2004 – Nov 2004	Praktikum bei der "Wincor-Nixdorf International GmbH", Abteilung Softwarequalität, Bereich Einzelhandel, Hamburg, Deutschland
Apr 2005 – Apr 2007	Studium des IT-Systems Engineering am "Hasso-Plattner-Institut" für Softwaresystemtechnik an der Universität Potsdam, Deutschland; Abschluss als Master of Science
Apr 2005 – Apr 2007	Stipendium des Deutschen Akademischen Austausch Dienst (DAAD)
Sep 2005 – Feb 2006	Studentische Hilfskraft am Lehrstuhl "Business Process Technology" (Prof. Dr. Mathias Weske), "Hasso-Plattner-Institut" für Softwaresystemtechnik an der Universität Potsdam, Deutschland
Mar 2007 – Sep 2007	Praktikum bei "SAP Labs", Bereich "imagineering", Palo Alto, USA
seit Okt 2007	Wissenschaftlicher Mitarbeiter und Doktorand am Lehrstuhl "Business Process Technology" (Prof. Dr. Mathias Weske), "Hasso-Plattner-Institut" für Softwaresystemtechnik an der Universität Potsdam, Deutschland
seit Okt 2007	Mitglied des Forschungskollegs "Service-Oriented Systems Engineering" am "Hasso-Plattner-Institut" für Softwaresystemtechnik an der Universität Potsdam, Deutschland