Parsing Behavior: The Hierarchical Nature of Concurrent Systems

Artem Polyvyanyy
Artem.Polyvyanyy@hpi.uni-potsdam.de

Behavioral models are the conceptual models that capture operational principles of real-world or designed systems. A behavioral model defines the state space of a system and the way the system can operate within its state space. A concurrent system allows for several threads of computation to execute simultaneously in the system. Parsing is a technique for discovering the structure of a behavioral model. The result of a parsing is a hierarchical decomposition of a model into logically independent units of behavior. In this paper, we report on two parsing techniques applicable for two different types of behavioral models. Sect. 1 discusses a technique for parsing workflow graphs, whereas Sect. 2 is devoted to parsing ordering relations. Finally, in Sect. 3, we sketch how these two parsing techniques can be related to provide a solution to the problem of structuring unstructured acyclic control flow specifications of concurrent systems under the behavioral equivalence notion which preserves the level of observable concurrency in the resulting structured model.

1 Parsing Workflow Graphs

Concurrent systems are often modeled using some kind of a directed flow graph, which we call a *workflow graph*, e.g., these are systems modeled in BPMN, EPC, UML activity diagrams, Petri nets, etc. A workflow graph can be parsed into a hierarchy of subgraphs with a single entry and single exit (SESE fragments, or fragments). Such a fragment can be addressed as a logically independent part of a concurrent system, in which the semantics of the fragment must be clarified based on the semantics of the respective modeling language. The result of the parsing procedure is a parse tree, which is the containment hierarchy of all fragments of a workflow graph.

The Refined Process Structure Tree (RPST) is a technique for workflow graph parsing which has various applications, e.g., translation between process languages, control-flow and data-flow analysis, process comparison and merging, process abstraction, process comprehension, model layout, and pattern application in process modeling. The RPST 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. Finally, the RPST of a workflow graph is the set of its canonical fragments, where a fragment is said to be canonical if it does not overlap on the set of edges with any other fragment of the graph.

In [17], we proposed an alternative way to compute the RPST that is simpler than the one developed originally [20]. In particular, the computation is reduced to constructing the *tree of the triconnected components* [5, 18] of a workflow graph in the special case when every node has at most one incoming or at most one outgoing edge.

A triconnected graph is a graph such that if any two nodes are removed from the graph, the resulting graph stays connected. A pair of nodes whose removal renders the graph disconnected is called a separation pair. Triconnected components of a graph are again graphs, smaller than the given one, that describe all separation pairs of the graph. Each triconnected component belongs to one out of four structural classes: A *trivial* (T) component consists of a single edge. A *polygon* (P) component represents a sequence of components. A *bond* (B) stands for a collection of components that share a common separation pair. Any other component is a *rigid* (R) component.

In this report, we only sketch the simplified procedure for construction of the RPST, whereas for the details we refer the reader to [17]. The simplified procedure for computing the RPST of a workflow graph can be summarized as follows: First, we normalize a workflow graph 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 normalized workflow graph, which coincides with its tree of the triconnected components, cf., Sect. 3.1 in [17]. Finally, we project the RPST of the normalized workflow graph onto the original graph and obtain its RPST.

Figure 1(a) shows a workflow graph and its triconnected components. Triconnected components are defined by dotted boxes, i.e., a triconnected component is composed of edges that are inside or cross the boundaries of the corresponding box. The workflow graph in Figure 1(a) is composed of two non-trivial triconnected components: P1 and B1. Note that names of components hint at their structural class. Figure 1(b) shows the tree of the triconnected components of the graph in Figure 1(a), which is an alternative representation of all triconnected components of a graph. Each node of the tree represents a triconnected component that is composed of components that are its descendants in the tree.

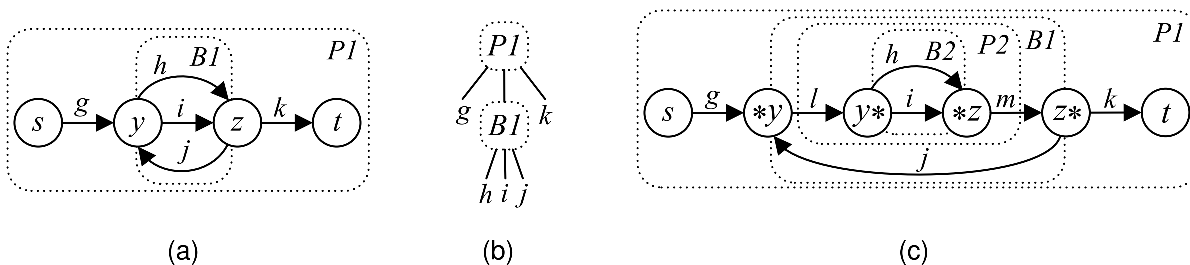

Figure 1: (a) A workflow graph and its triconnected component subgraphs, (b) the tree of the triconnected components of (a), and (c) the normalized version of (a) and its triconnected component subgraphs

Figure 1 demonstrates the concept of node-splitting. If the splitting is applied to node y of the graph in Figure 1(a), it results in the new graph given in Figure 1(c) with three fresh elements: nodes $*y$ and $y*$, and edge l. After splitting nodes y and z in the graph in Figure 1(a), the graph in Figure 1(c) is a normalized version of the graph in Figure 1(a).

After normalization, the simplified algorithm for constructing the RPST proceeds by computing the tree of the triconnected components of the normalized graph. This tree coincides with the RPST of the normalized graph, cf., [17]. Next, this tree must be projected onto the original graph by deleting all the edges introduced during node-splittings. 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. The final stage of the algorithm for computing the RPST of the workflow graph in Figure 1(a) is exemplified in Figure 2.

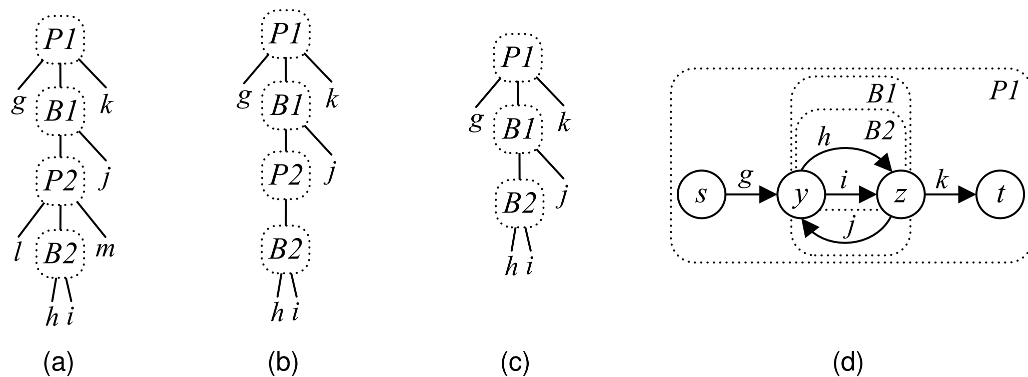

Figure 2: (a) The tree of the triconnected components of the workflow graph in Figure 1(c), (b) the tree from (a) without the fresh edges l and m, (c) the RPST of the workflow graph in Figure 1(a), and (d) the workflow graph from Figure 1(a) and its canonical fragments

The tree of the triconnected components of the normalized graph, cf., Figure 1(c), consists of four triconnected components: $P1$, $B1$, $P2$, and $B2$. Figure 2(a) shows the corresponding tree of the triconnected components. One can see the RPST without trivial fragments that correspond to the fresh edges l and m in Figure 2(b). Observe that $P2$ now specifies the same set of edges as $B2$. Therefore, we omit $P2$, which is redundant, to obtain the tree given in Figure 2(c). This tree is the RPST of the original graph that is given in Figure 1(a). Finally, Figure 2(d) visualizes the graph again together with its canonical fragments. In comparison with the triconnected decomposition shown in Figure 1(a) and Figure 1(b), by following the described procedure we additionally discovered canonical fragment $B2$. $P1$, $B1$, and $B2$ are all the canonical fragments of the workflow graph. For the proof of the fact that resulting tree is indeed the RPST of the original graph we refer the reader to [16].

2 Parsing Ordering Relations

Concurrent systems can be described with the help of ordering relations between pairs of tasks or pairs of occurrences of tasks. There exist different notions of ordering relations, e.g., unfolding relations, cf., [4, 11, 12], behavioral profile [21], relations of the α mining algorithm [19], etc. These relations are, essentially, behavioral abstractions

that capture core behavioral characteristics of a system at different levels of detail. Examples for such behavioral characteristics are causality, conflict, and concurrency. In this section, we discuss a technique of parsing ordering relations that can be applied to any given notion of ordering relations. The parsing decomposes ordering relations into *clans*, each with clear behavioral characteristics specific to the employed notion of ordering relations. To make parsing possible, we give a structural characterization to ordering relations, i.e., ordering relations are treated as a generalization of a directed graph.

The adjacency array representation of a directed graph $D = (V, E)$ is a coloring of a set $E_2(V) = \{(v_1, v_2) \mid v_1, v_2 \in V,\ v_1 \neq v_2\}$, where $E \subseteq E_2(V)$, with two colors, e.g., 0 and 1. Therefore, an adjacency array of a directed graph can be given by an indicator function $I_E : E_2(V) \to \{0, 1\}$. The notion of a *two-structure* is a generalization of the notion of a graph [3]. A two-structure allows an arbitrary coloring of the set $E_2(V)$. A *two-structure* is an ordered pair $S = (N, R)$ such that N is a nonempty finite set of nodes, and R is an equivalence relation on $E_2(N)$.

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 $e_1\ R\ e_2$, if and only if $\alpha(e_1) = \alpha(e_2)$; C is a set of colors. Observe that a coloring function α is not unique, as the choice of colors can be arbitrary.

Given the ordering relations, we treat them as a two-structure where nodes are tasks, over which relations are defined, and colors of edges encode different types of relations. An equivalence class of the equivalence relation of such a two-structure represents all ordering relations of the same type, e.g., causality.

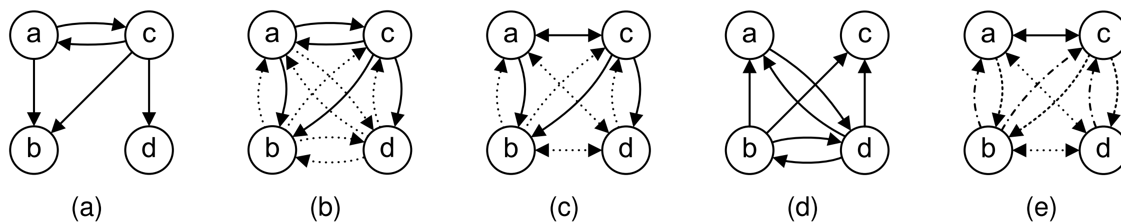

Figure 3: (a),(d) Directed graphs, and (b),(c),(e) two-structures

A directed graph that is defined by the pair (V, E), where $V = \{a, b, c, d\}$ and $E = \{(a, c), (c, a), (a, b), (c, b), (c, d)\}$, is shown in Figure 3(a), whereas Figure 3(b) presents one of the possible corresponding two-structures (N, R). The two-structure has two equivalence classes of edges, where one class contains edges E (drawn with solid edges) and the other one contains edges $E_2(N) \setminus E$ (drawn with dotted edges). Figure 3(c) shows the same two-structure using a simplified notation, i.e., symmetric edges are drawn as two-sided arrows. Notice that the correspondence between the two-structure and the graph is rather arbitrary, as one can also accept the two-structure as such that corresponds to the graph in Figure 3(d) by exchanging the roles of its equivalence classes. Alternatively, one can define a correspondence by using larger sets of colors, e.g., the 2-structure given in Figure 3(e) uses four equivalence classes.

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

$S = (N, R)$ be a two-structure. A node $n \in N$ *distinguishes* nodes $m, k \in N$, if and only if (n, m) and (n, k) are of different colors or (m, n) and (k, n) are of different colors. 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 $(x, z)\ R\ (y, z)$.

Let $S = (N, R)$ with $|N| > 1$ and P be a partition of $E_2(N)$ induced by R. It follows immediately that $\emptyset$, N, and the singletons $\{n\}$, $n \in N$, are clans of S. These clans are the *trivial* clans of S. S is complete, if and only if $|P| = 1$. S is linear, if and only if $|P| = 2$ 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 equivalence class of R and the edges $\{(n_j, n_i) \mid i < j\}$ form an equivalence class of R. S is *primitive*, if and only if it contains at least three nodes and all clans in S are trivial.

Construction principles of a two-structure are defined by its decomposition into factors and a quotient that gives the relations between the factors. Let $S = (N, R)$ be a two-structure. A partition $\chi = \{X_1, \ldots, X_k\}$ 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 $(X_1, Y_1)\ R_\chi\ (X_2, Y_2)$, if and only if $(x_1, y_1)\ R\ (x_2, y_2)$ 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_i} with respect to a factorization $\chi = \{X_1, \ldots, X_k\}$ and the quotient S/χ.

A nonempty clan X of S is *prime*, if and only if for all clans Y of S holds that X and Y do not overlap. We denote by $\mathcal{C}(S)$ the set of all clans of S. 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 N. We denote by $\mathcal{P}_{max}(S)$ the set of all maximal prime clans of S; if $|N| = 1$, then $\mathcal{P}_{max}(S) = \{N\}$.

The maximal prime clans $\mathcal{P}_{max}(S)$ of a two-structure S form a partition of N, i.e., the domain of each two-structure can be partitioned by the domains of its maximal prime clans. For each two-structure S, the quotient $S/\mathcal{P}_{max}(S)$ is either primitive, or complete, or linear, cf., [3].

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.

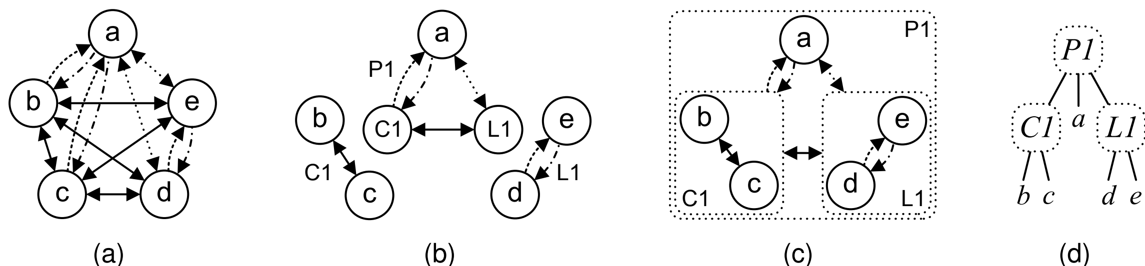

Figure 4: (a) A two-structure, (b) clans of (a), and (c),(d) the hierarchy of clans of (a)

Figure 4 exemplifies the decomposition of a two-structure. Figure 4(a) shows a two-structure which is composed of five nodes and has four equivalence classes on edges.

Partition $\chi = \{\{a\},\{b,c\},\{d,e\}\}$ is factorization of this two-structure. Two-structures induced by subsets of nodes $\{a\}$, $\{b,c\}$, and $\{d,e\}$ are, respectively, a trivial, a complete, and a linear clan of the original two-structure. Clan $P1$ (of class primitive), cf., Figure 4(b), is the quotient of the two-structure by factorization χ; observe that clan names hint at their class. Finally, Figure 4(c) organizes clans in a hierarchy; each quotient and each nontrivial clan is enclosed in a dotted box with rounded corners, whereas containment of boxes represents the parent-child relation of quotients and clans. Figure 4(d) shows a tree representation of the decomposition.

3 Structuring Acyclic Concurrent Systems

Concurrent systems modeled as graphs can have almost any topology. However, it is often preferable that they follow some structure. In this respect, a well-known property of concurrent systems is that of *(well-)structuredness* [6], meaning that for every node with multiple outgoing arcs (a *split*), there is a corresponding node with multiple incoming arcs (a *join*), such that the set of nodes between the split and the join form a SESE fragment. For example, Figure 5(a) shows an unstructured system, while Figure 5(b) shows an equivalent structured system. Note that Figure 5(b) uses short-names for tasks (a, b, c . . .), which appear next to each task in Figure 5(a). We assume a simple modeling language, i.e., a concurrent system is composed of tasks, events, gateways, and sequence flow edges. We allow exclusive and parallel gateways. Our modeling language can be seen as a basic subset of BPMN.

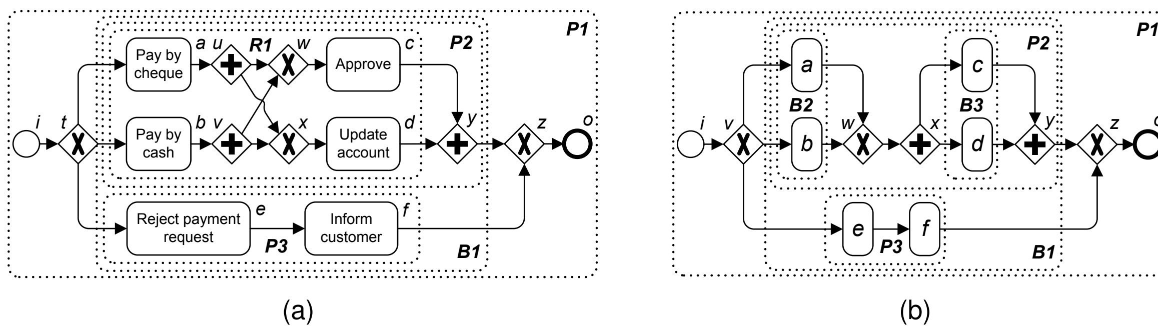

Figure 5: (a) Unstructured concurrent system and (b) its equivalent structured version

This section sketches the main idea of the solution to the problem of automatically transforming acyclic concurrent systems, whereas the details can be found in [15]. The motivations for such a transformation are manifold. Firstly, it has been empirically shown that structured models are easier to comprehend and less error-prone than unstructured ones [8]. Thus, a transformation from an unstructured to a structured system can be used as a refactoring technique to increase model understandability. Secondly, a number of existing analysis techniques only work for structured systems [2, 7]. By transforming unstructured models into structured ones, we can extend the applicability of these techniques to a larger class of models. Thirdly, a transformation from unstructured

to structured models can be used to implement converters from graph-oriented process modeling languages to structured process modeling languages, e.g., transforming from BPMN models to BPEL executable code.

As mentioned above, the problem of structuring concurrent systems is relevant in the context of designing BPMN-to-BPEL transformations. However, BPMN-to-BPEL transformations, such as [14], treat rigids as black-boxes that are translated using BPEL links or event handlers, rather than seeking to structure them. A large body of work on flowcharts and GOTO program transformation [13] has addressed the problem of structuring rigid fragments composed of exclusive gateways. In some cases, these transformations introduce additional boolean variables in order to encode part of the control flow, while in other cases they require certain nodes to be duplicated. In [6], the authors show that not all acyclic rigids composed of parallel gateways can be structured. They do so by providing one counter-example, but do not give a full characterization of the class of models that can be structured nor do they define any automated transformation. Instead, they explore some causes of unstructuredness. In a similar vein, [9] presents a taxonomy of unstructuredness in process models, covering cyclic and acyclic rigids. However, the taxonomy is incomplete, i.e., it does not cover all possible cases of models that can be structured. Also, the authors do not define an automated structuring algorithm.

The RPST of a well-structured system contains no rigid fragments. If one could transform each rigid fragment into an equivalent structured fragment, the entire model could be structured by traversing the RPST bottom-up and replacing each rigid by its equivalent structured fragment. Observe that in Figure 5, the only rigid fragment R1 in Figure 5(a) is replaced by an equivalent polygon fragment P2 in Figure 5(b).

Our goal is that the structured system preserves the level of observable concurrency of the equivalent unstructured system, i.e., we require that both systems are fully concurrent bisimilar [1]. The core idea of the structuring method proposed in [15] is to compute the ordering relations, in particular the unfolding relations [4, 11, 12], of every rigid fragment, and to synthesize a structured fragment from these ordering relations (if such a structured fragment exists). To this end, the unfolding relations are computed on the alternative representation of a system, viz. its complete prefix unfolding [11]. An unfolding is a “compact” representation of all concurrent runs (instance subgraphs) of a system. A complete prefix unfolding is a part of the unfolding that contains information about all states that are reachable by the system.

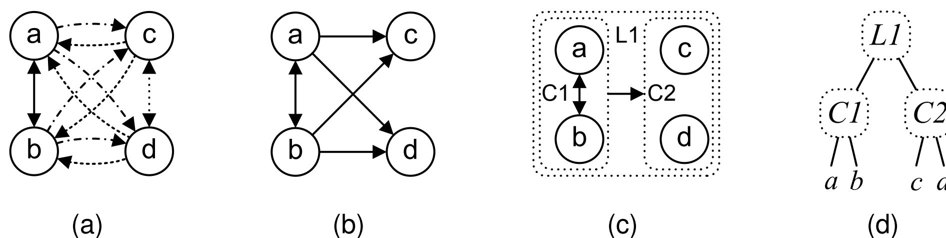

Figure 6: Ordering relations of systems in Figure 5(a) and Figure 5(b) given (a) as a two-structure and (b) as a directed graph. (c) Modular decomposition of (b) and (d) its tree representation.

For the technical details on computing unfolding relations we refer the reader to [15]. A two-structure in Figure 6(a) shows the unfolding relations computed for fragment R1 of the system in Figure 5(a). The equivalence relation contains four equivalence classes that represent causality, inverse causality, conflict, and concurrency relations. Figure 6(a) must be read as follows: Solid edges represent the conflict relation ($\{(a,b),(b,a)\}$). Dotted edges stand for the concurrency relation ($\{(c,d),(d,c)\}$). The causality relation is encoded by dash dotted lines ($\{(a,c),(a,d),(b,c),(b,d)\}$). Finally, the inverse causality relation is given by dashed lines ($\{(c,a),(d,a),(c,b),(d,b)\}$). Because of the nature of unfolding relations, the corresponding two-structure can always be represented by an equivalent directed graph, cf., Figure 6(b). We call such a graph the *ordering relations graph*. In this graph, two-sided arrows hint at conflict, absence of an edge between a pair of nodes signals for concurrency, and a directed edge stands for causality.

The important observation in the context of the structuring problem is that the system in Figure 5(b) also exposes unfolding relations that can be represented by the ordering relations graph in Figure 6(b) [15]. However, the well-structured system is not given, rather it needs to be synthesized from the ordering relations graph. To this end, we employ the technique for parsing ordering relations, cf., Sect. 2. Decomposition of a directed graph into clans is known as modular decomposition and can be accomplished in linear time [10]. Figure 6(c) shows the decomposition, whereas Figure 6(d) gives its tree representation.

Finally, we conclude that there exists an equivalent well-structured system, if and only if decomposition of the ordering relations graph of the unstructured system contains no primitive clan, cf., [15]. A complete clan can be represented as a bond fragment as all nodes of a complete clan are pairwise in the same ordering relation. If this relation is the conflict relation, then one can construct a bond with exclusive gateways; in the case of the concurrency relation, on the other hand, one can construct a bond with parallel gateways. A linear clan can be represented by a polygon fragment in the resulting well-structured fragment. Therefore, in order to construct a well-structured fragment, one needs to traverse the hierarchy of clans bottom-up and synthesize a bond fragment for each complete clan and a polygon fragment for each linear clan. For instance, complete clan C1 in Figure 6(d) corresponds to bond B2 in Figure 5(b), C2 corresponds to B3, and $L1$ corresponds to P2.

4 Conclusion

In this report, we have discussed two techniques for parsing two different representations of concurrent systems. Parsing can be used to learn the hierarchical structure of a concurrent system. First, we sketched the simplified algorithm for computing the Refined Process Structure Tree—a technique for workflow graph parsing. Second, we discussed a technique that can be used to parse concurrent systems specified as ordering relations between pairs of tasks or pairs of occurrences of tasks. Finally, we showed how these two techniques relate to each other in a solution to the problem of structuring acyclic concurrent systems.

References

[1] Eike Best, Raymond R. Devillers, Astrid Kiehn, and Lucia Pomello. Concurrent bisimulations in petri nets. *Acta Informatica*, 28(3):231–264, 1991.

[2] Carlo Combi and Roberto Posenato. Controllability in temporal conceptual workflow schemata. In *BPM*, volume 5701 of *LNCS*, pages 64–79. Springer, 2009.

[3] Andrzej Ehrenfeucht and Grzegorz Rozenberg. Theory of 2-structures, Part I: Clans, basic subclasses, and morphisms. *Theoretical Computer Science (TCS)*, 70(3):277–303, 1990.

[4] Joost Engelfriet. Branching processes of petri nets. *Acta Informatica*, 28(6):575–591, 1991.

[5] John E. Hopcroft and Robert Endre Tarjan. Dividing a graph into triconnected components. *SIAM Journal on Computing (SIAMCOMP)*, 2(3):135–158, 1973.

[6] Bartek Kiepuszewski, Arthur H. M. ter Hofstede, and Christoph Bussler. On structured workflow modelling. In *CAiSE*, volume 1789 of *LNCS*, pages 431–445. Springer, 2000.

[7] Manuel Laguna and Johan Marklund. *Business Process Modeling, Simulation, and Design*. Prentice Hall, 2005.

[8] Ralf Laue and Jan Mendling. The impact of structuredness on error probability of process models. In *UNISCON*, volume 5 of *LNBIP*, pages 585–590. Springer, 2008.

[9] Rong Liu and Akhil Kumar. An analysis and taxonomy of unstructured workflows. In *BPM*, volume 3649 of *LNCS*, pages 268–284. Springer, 2005.

[10] Ross M. McConnell and Fabien de Montgolfier. Linear-time modular decomposition of directed graphs. *Discrete Applied Mathematics*, 145(2):198–209, 2005.

[11] Kenneth L. McMillan. A technique of state space search based on unfolding. *Formal Methods in System Design (FMSD)*, 6(1):45–65, 1995.

[12] Mogens Nielsen, Gordon D. Plotkin, and Glynn Winskel. Petri nets, event structures and domains. In *Semantics of Concurrent Computation*, volume 70 of *LNCS*, pages 266–284. Springer, 1979.

[13] G. Oulsnam. Unravelling unstructured programs. *The Computer Journal (CJ)*, 25(3):379–387, 1982.

[14] Chun Ouyang, Marlon Dumas, W. M. P. van der Aalst, Arthur H. M. ter Hofstede, and Jan Mendling. From business process models to process-oriented software systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 19(1), 2009.

[15] Artem Polyvyanyy, Luciano García-Bañuelos, and Marlon Dumas. Structuring acyclic process models. In *BPM*, volume 6336 of *LNCS*, pages 276–293. Springer, 2010.

[16] Artem Polyvyanyy, Jussi Vanhatalo, and Hagen Völzer. Simplified computation and generalization of the refined process structure tree. Technical Report RZ 3745, IBM, 2009.

[17] Artem Polyvyanyy, Jussi Vanhatalo, and Hagen Völzer. Simplified computation and generalization of the refined process structure tree. In *WS-FM*, 2010. to appear.

[18] Robert Endre Tarjan and Jacobo Valdes. Prime subprogram parsing of a program. In *POPL*, pages 95–105. ACM, 1980.

[19] Wil M. P. van der Aalst, Ton Weijters, and Laura Maruster. Workflow mining: Discovering process models from event logs. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 16(9):1128–1142, 2004.

[20] Jussi Vanhatalo, Hagen Völzer, and Jana Koehler. The refined process structure tree. *Data and Knowledge Engineering (DKE)*, 68(9):793–818, 2009.

[21] Matthias Weidlich, Artem Polyvyanyy, Jan Mendling, and Mathias Weske. Efficient computation of causal behavioural profiles using structural decomposition. In *Petri Nets*, volume 6128 of *LNCS*, pages 63–83. Springer, 2010.